2017

Ensuring the Observability of Structural Test Obligations

Ying Meng
University of South Carolina

Ensuring the Observability of Structural Test Obligations

by

Ying Meng

Bachelor of Science
Xidian University 2001

Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science in

Computer Science

College of Engineering and Computing

University of South Carolina

2017

Accepted by:

Gregory Gay, Director of Thesis

Srihari Nelakuditi, Reader

Ioannis Rekleitis, Reader

Cheryl L. Addy, Vice Provost and Dean of the Graduate School

Acknowledgments

I would like to thank all the people who have made this thesis possible. My deepest gratitude is to my advisor Dr. Gregory Gay for his invaluable supportand guidance of my research and for instilling in me the qualities of being a researcher. He has the attitude and the substance of a great advisor: he continually and convincingly conveyed a spirit of adventure in regard to research. Without his guidance and persistent help this thesis would not have been possible.

I am deeply grateful to my committee members Dr. Srihari Nelakuditi, and Dr. Ioannis Rekleitis for serving as my committee members even at hardship. I also want to thank you for invaluable comments and suggestions to my thesis, defense, and my study.

I also owe a great debt of gratitude to Dr. Manton M. Matthews for your assistance and guidance throughout my graduate study over years.

I am thankful to the Advanced Technology Center at Rockwell Collins Inc. for granting access, through Michael Whalen, to industrial case examples.

I am grateful to all my friends Hussein Almulla, Alireza Salahirad, Srujana Bollina, Harui Wu, Li Wang, Lihong Dai, Shan Zhu, and Yixuan Liu (too many to list here but you know who you are!) for being supportive throughout my time here.

A special thanks to my dear family. Words cannot express how grateful I am to my parents for all of the sacrifices that you've made on my behalf. I would like to thank my sister and my brother, you have been my best friends all my life. I would also like to thank to my beloved husband Xinhua Liu and son Siyou Liu. Thank you for always supporting, encouraging, and loving me. I love all of you.

Abstract

Test adequacy criteria are widely used to guide test creation. However, many of these criteria are sensitive to statement structure or the choice of test oracle. This is because such criteria ensure that execution *reaches* the element of interest, but impose no constraints on the execution path *after* this point. We are not guaranteed to observe a failure just because a fault is triggered. To address this issue, we have proposed the concept of *observability*—an extension to coverage criteria based on Boolean expressions that combines the obligations of a host criterion with an additional path condition that increases the likelihood that a fault encountered will propagate to a monitored variable.

Our study, conducted over five industrial systems and an additional forty open-source systems, has revealed that adding observability tends to improve efficacy over satisfaction of the traditional criteria, with average improvements of up to 392.44% in mutation detection and per-model improvements of up to 1654.38%. Ultimately, there is merit to our hypothesis—observability reduces sensitivity to the choice of oracle and to the program structure.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Test adequacy criteria defined over program structures—such as statement, branches, or atomic conditions—are widely used as measures to assess the efficacy of test suites. Such criteria are essential and offering guidance in the testing process, as they offer clear checklists of goals to testers and the means to automate the creation of test suites. However, many of these criteria are highly sensitive to how statements are structured [23, 50] or the choice of test oracle [24, 59, 61].

Consider the Modified Condition/Decision Coverage (MC/DC) coverage criterion [8]. MC/DC is used as an exit criterion when testing software for critical software in the avionics domain. For certification of such software, a vendor must demonstrate that the test suite provides MC/DC coverage of the source code [54]. However, the efficacy of test suites created to satisfy MC/DC—particularly when test suite creation is automated—is highly dependent on the syntactic structure of the code under test. A complex Boolean expression, for example, could be written as a series of simple expressions, or as a single *inlined* expression. This simple transformation can dramatically improve the efficacy of MC/DC-satisfying test suites, with increases in fault detection of up to 4542.47% [23].

Such results are worrying, particularly given the importance of coverage criteria in safety certification, and the improvements made in terms of automated generation. When examining the discrepancy in efficacy between test suites for non-inlined and inlined programs, we often found that the test case encountered a fault in the code—such as an erroneous Boolean operator—leading to a corrupted internal state.

However, this corruption was *masked* out in a subsequent expression, and did not propagate to an output. This effect was far more prevalent in programs with many simple Boolean expressions, whose results were stored in intermediate values. In addition, regardless of structure, it was common that a test case encountered a fault, leading to a corrupted internal state, but the test case did not allow sufficient execution time for the corrupted state to propagate to an output; the test case terminated before the corrupted state became visible in a variable monitored by the test oracle.

This sensitivity to structure and choice of oracle is caused by the fact that the obligations for structural coverage criteria are only posed over specific syntactic elements—statements, branches, conditions. Such obligations ensure that execution *reaches* the element of interest, and exercises it in the prescribed manner. However, no constraints are imposed on the execution path *after* this point. We are not guaranteed to observe a failure just because a fault is triggered.

To address this issue, we have proposed the concept of *observability*—an extension to coverage criteria based on Boolean expressions that has the potential to eliminate masking. Observable coverage criteria combine the test obligations of their host criterion with an additional path condition that increases the likelihood that a fault encountered when executing the element of interest will propagate to a variable monitored by the test oracle. Unlike many extensions to coverage criteria [56], this path condition does not increase the number of test obligations over its host criterion. Instead, it makes the existing obligations more stringent to satisfy, as the possibility of propagating a fault revealed by the original obligation must also be demonstrated. We hypothesize that this additional observability constraint will improve the effectiveness of the host criterion—no matter which criterion is chosen—particularly when used as a test generation target, paired with common output-based test oracles.

This work is an extension of our prior work defining and exploring the concept of observability [65, 25, 67]. We first proposed the concept of observability as an

extension of the MC/DC coverage criterion [65]. An extended study found that that OMC/DC was more effective—and overcame many of the weaknesses of—traditional coverage criteria for a small set of studied industrial systems [25].

This work extends previous efforts by decoupling the notion of observability from MC/DC and exploring its application as a generic addition to any coverage criterion. This decoupling allows us to explore the impact of the choice of host criterion, and to explore the efficacy of observability as a general extension to adequacy criteria. Our new experimental studies also consider a far wider range of programs than previously explored in order to better understand the efficacy of observability-based coverage criteria when used as the target of automated generation.

Our study, conducted over five industrial systems from Rockwell Collins and an additional forty open-source systems, has revealed the following insights:

- Test suites satisfying Observable MC/DC are generally the most effective, killing 95.61% of mutants on average (MX oracle) and 87.03% (OO oracle) for the inlined Rockwell models, 98.85% (MX)/85.88% (OO) for the non-inlined Rockwell models, and 89.62% (MX)/65.14% (OO) for the Benchmarks models.

- Adding observability tends to improve efficacy over satisfaction of the traditional criteria, with average improvements of up to 392.44% in mutation detection and per-model improvements of up to 1654.38%.

- Factors that can harm efficacy—generally resulting in a reduction in the number of fulfilled obligations—include expression complexity, the length of the combinatorial path from expression to output, and the length of the delayed path from expression to output.

- The addition of observability results in an increase in the size of test suites. The magnitude of that increase depends on the length of the path from each expression to the output.

- The addition of observability results in an decrease in the number of fulfilled obligations. This loss is due to either the discovery of dead code that cannot influence the output or obligations that are too complex for the test generator to solve.

- The choice of host criterion influences the final efficacy, but the largest increase in complexity comes from the addition of observability itself. Varying both dimensions—criterion and observability—may allow testers to find an optimal level of efficacy and complexity.

- Observability reduces sensitivity to the choice of oracle, by ensuring a masking-free path from expression to the variables monitored by the test oracle.

- Observability reduces sensitivity to the program structure by capturing the complexity benefits of inlining in the path from expression to output.

Based on our results, *observability* is a valuable extension—regardless of the chosen host criterion. The addition of observability increases test efficacy and produces test suites that are robust to changes in the structure of program or the variables under monitored by test oracle.

The remainder of this thesis is structured as follows. Chapter 2 introduces important background material. Chapter 3 presents the concept of observability and offers formal definitions and implementation details. Chapter 4 presents the details of our experiments, and Chapter 5 discusses our observations. Chapter 6 presents related work. Finally, Chapter 7 summarizes and concludes the thesis.

Chapter 2

Background

In this research, we are interested in improvements to the criteria used to judge the adequacy of testing efforts—and to automatically generate test suites. In particular, we are focused on the safety-critical reactive systems that power our society. In this chapter, we will discuss background material on both topics.

2.1 Adequacy Criteria

The concept of adequacy is important in providing developers with the guidance needed to test effectively. As we cannot know what faults exist without verification, and as testing cannot—except in simple cases—conclusively prove the absence of faults, a suitable approximation must be used to measure the adequacy of our testing efforts. If existing tests have not surfaced any faults, is the software correct, or are the tests *inadequate*?

The most common methods of measuring adequacy involve coverage of structural elements of the software, such as individual statements, branches of control, and complex boolean expressions [38, 48, 49]. Each adequacy criterion prescribes requirements tests must fulfill. For example, branch coverage requires that all outcomes of expressions that can result in different code segments being executed—such as if-then-else and loop conditions—be executed. The idea of measuring adequacy through coverage is simple, but compelling: unless code is executed, many faults are unlikely to be found. If tests execute elements as prescribed by the criterion, than testing is deemed "adequate" with respect to faults that manifest through such structures.

Adequacy criteria have seen widespread use, as they offer objective, measurable checklists [26] and—importantly—*stopping criteria* for the testing process. For that same reason, they are ideal as test generation targets [52, 44, 35], as coverage can be straightforwardly measured and optimized for [5].

2.2 Structural Coverage

Structural coverage criteria serve as a means to determine that the structure of system under test—the various elements making up the source code—have been thoroughly exercised by test cases. Many structural coverage criteria, defined with respect to specific syntactic elements of a program, have been proposed and studied over the past decades [25, 21]. These have been used to measure suite adequacy—as a means to assess the quality of existing test suites, and whether developers can stop adding tests. They are also commonly used as as targets for automated test generation.

In this study, we are primarily concerned with reactive systems—safety-critical embedded systems that interact with the physical world. Such systems often have sophisticated logical structures in the code. Therefore, in this work, we are primarily concerned with structural coverage criteria defined over Boolean expressions. In particular, we are focused on Condition Coverage, Branch Coverage, Decision Coverage, and Modified Condition/Decision Coverage (MC/DC).

Decision Coverage A decision is any Boolean expression within the program. Decisions are composed of one or more conditions—atomic Boolean subexpressions—connected by operators (`and`, `or`, `xor`, `not`). Decision Coverage requires that all decisions in the system under test evaluate to both the `true` and `false`. Given the expression `((a and b) and (not c or d))`, tests would need to be produced where the expression evaluates to true and the expression evaluated to false. In this case, the test input `(TTTT),(TTTF)` would achieve Decision Coverage.

Branch Coverage A branch is a particular type of decision that can cause program execution to diverge down a particular control flow path, such as that in `if` or `case` statements. Branch Coverage is defined in the same manner as Decision Coverage, but is restricted to branches, rather than all decision statements. Branch Coverage is arguably the most commonly used coverage criterion, with ample tool support[1] and industrial adoption. Improving Branch Coverage is a common goal in automated test generation [44, 15].

Condition Coverage A condition is an atomic Boolean subexpression within the broader decision. Condition Coverage requires that each condition evaluate to `true` and `false`. Given the expression `((a and b) and (not c or d))`, achieving Condition Coverage requires tests where the individual atomic Boolean conditions `a`, `b`, `c`, and `d` evaluate to `true` and `false`. For this decision, test input `(TTTF),(FFFT)` would achieve Condition Coverage.

Note that achieving one form of coverage does not always imply that others are fulfilled as well. The test input given above would achieve Condition Coverage, but not Decision Coverage, as both test inputs result in the decision evaluating to `false`. Similarly, the input provided earlier for Decision Coverage—`(TTTT),(TTTF)`—would not achieve Condition Coverage, as only `d` evaluates to both outcomes. Therefore, stronger criteria—such as Modified Condition/Decision Coverage—require that the obligations of both Decision and Condition Coverage be met.

Modified Condition/Decision Coverage (MC/DC) Criterion The MC/DC criterion is used as an exit criterion when testing software for critical software in the avionics domain, and is required for safety certification in that domain [55]. MC/DC further strengthens Condition and Decision Coverage by requiring that each decision

[1]Such as the popular Cobertura and EMMA IDE plug-ins—see `http://cobertura.github.io/cobertura/` and `http://www.eclemma.org/`

evaluate to all possible outcomes, each condition take on all possible outcomes, and that each condition within a decision be shown to independently impact the outcome of the decision.

Independent effect is defined in terms of *masking*, which means that the condition has no effect on the value of the decision as a whole; for example, given a decision of the form `x and y`, the truth value of `x` is irrelevant if `y` is false, so we state that `x` is masked out. A condition that is not masked out has *independent effect* for the decision.

Consider again the expression `((a and b) and (not c or d))`. Suppose we examine the independent affect of `d` in the example; if `(a and b)` evaluates to false, than the entire decision will evaluate to false, masking the effect of `d`; Similarly, if `c` evaluates to false, then `(not c or d)` evaluates to true regardless of the value of `d`. Only if we assign `a`, `b`, and `c` the value of true does the value of `d` affect the outcome of the decision. Showing independent impact requires a pair of test cases where all other conditions hold fixed values and our condition of interest flips values. If changing the value of the condition of interest changes the value of the decision as a whole, then the independent impact has been shown. In this example, the test inputs `(TTTT), (TTTF), (FTTT), (TFTT), and (TTFF)` satisfies MC/DC. Tests inputs 1 and 3 show the effect of `a`, 1 and 4 show `b`, 2 and 5 show `c`, and 1 and 2 show `d`. MC/DC can be achieved in $(number\ of\ conditions + 1)$ test cases if care is taken in selecting test input.

Because both decisions and conditions are covered, we state that MC/DC *subsumes* the previously-defined forms of coverage. Achieving MC/DC also achieves Decision and Condition Coverage. This comes at a cost—satisfying MC/DC requires more test cases and more effort than satisfying any of the above criteria. Therefore, if no benefit is perceived from the additional requirements of MC/DC, testers often elect to satisfy a simpler criterion instead.

Several variations of MC/DC exist—for this study, we use Masking MC/DC, as it is a common criterion within the avionics community [7].

2.3 Reactive Systems and Dataflow Languages

Increasingly, our society is powered by sophisticated software systems—such systems manage factories and power plants, coordinate the many systems driving automobiles and airplanes, and even make life-saving decisions as part of medical devices implanted in human bodies. Many of these systems are what we refer to as *reactive systems*—-embedded systems that interact with physical processes. Reactive systems operate in cycles—receiving new input from their environment, to which they react by issuing output.

Such systems are commonly designed using modeling languages, which are translated into C code that can be directly flashed to the embedded hardware. Models can be developed using visual notations, such as Simulink [42], Stateflow [43] and SCADE [11]. They can also be directly expressed using *dataflow languages*, such as Lustre.

Lustre is a synchronous dataflow language used in a number of domains to model or directly implement embedded systems [29]. It is a declarative programming language for manipulating *data flows*—infinite streams of variable values. These variables correspond to traditional data types, such as integers, booleans, and floating point numbers. Lustre offers an intermediate representation between behavioral model and traditional source code that is useful for specification, design, and analysis purposes. Because of the simplicity and declarative nature of Lustre, it is well-suited to model checking and verification, in particular with regards to its safety properties [27]. Lustre programs can be automatically generated from visual notations such as Simulink, and can be automatically compiled to target languages such as C/C++, VHDL, as well as to input models for verification tools such as model checkers.

A Lustre program is structured into a network of control modules (nodes) that specify relations between inputs and outputs of a system. A node is the specification of a stream transformer, mapping the streams of input variables to the streams of internal and output variables using a set of defined expressions. Lustre nodes have cyclic behavior—at execution cycle i, the node takes in the values of the input streams at instant i, manipulates those values, and issues new values for the internal and output variables. Nodes have a limited form of memory, and can access input, internal, and output values from previous instants (up to a statically-determined finite limit). To update program state within one computational step, combinatorial variables are used; to store current program state for the reference by later cycle or cycles, delay variables are used (i.e., $\frac{1}{z}$ blocks in Simulink). During a cycle, all variables are calculated according to their definitions: combinatorial variables are computed combinatorially using values at the current computational step, and delay variables are computed combinatorially using values from previous step or steps.

The body of a Lustre node consists of a set of equations of the form $x = expr$ where x is a variable identifier, and t is the expression defining the value of x at instant i. Like in most programming languages, expression t can make use of any of the other input, internal, or output variables in defining x—as long as that variable has already been assigned a value during the current cycle of computation. The order of equations does not matter in Lustre, except for data dependencies. That is, within a computational step, as long as all the variables involved in an equation have already been computed, the equation can be evaluated.

Lustre supports many of the traditional numerical and boolean operators, including $+$, $-$, $*$, $/$, $<$, $>$, $\%$, etc. Lustre also supports two important *temporal* operators: $pre(x)$ and $\rightarrow$. The $pre(x)$ operator, or "previous", evaluates to the value of x at instant $(i-1)$. The $\rightarrow$ operator, or "followed by", allows initialization of variables in the first instant of execution. For example, the expression $x = 0 \rightarrow pre(x) + 1$ defines

the value of x to be 0 in instant 0, then defines it as 1 at instant 1—or, the value at instant 0 plus one—and so forth.

```
v1 = (0 -> (pre in1));
v2 = (v1 > 1);
v3 = (false -> (pre v2));
out = (if in2 then v2 else v3);
```

Figure 2.1: Sample Lustre Code Fragment

For example, given following code fragment 2.1, in which *in1* and *in2* are input variables, *v1*, *v2*, and *v3* are internal variables, and *out* is an output variable. Variables *in1* and *v1* are type of int and all the rests are type of boolean. Variables *in1* and *v2* are delay variables, values stored in them will be used by *v1* and *v3* in the next cycle, respectively. Variable *v1* is initially assigned to *0* followed by (represented by operator *arrow*) *in1*'s value from the previous cycle, at each subsequent cycle. Similarly, values of variable *v3* is a stream of boolean values, which starts with a *false* followed by *v2*'s value from the previous computational step.

2.3.1 Test Case Structure for Reactive Systems

There are two key artifacts necessary to construct a test case, the *test inputs, or test data*—inputs given to the system under test—and the *test oracle*—a judge on the resulting execution [32, 63]. A *test oracle* can be defined as a predicate on a sequence of stimuli to and reactions from the SUT that judges the resulting behavior according to some specification of correctness [6].

As reactive systems compute in cycles, multiple test inputs must generally be provided. Therefore, tests are divided into a series of *test steps*, where input and expected output is provided for *each step*. In each step, specific values are given for each input variable, then the internal and output variables are computed accordingly. The output at each step is compared to the expected output provided as part of the

test oracle. Table 2.1 shows example test input that contains four steps together with corresponding evaluations of all internal and output variables. From this example, we can see how the values of delay variables impact other variables.

Table 2.1: Sample Lustre Program Evaluation

step	inputs (in1, in2)	internals (v1, v2, v3)	outputs (out)
1	(1, T)	(0, F, F)	(F)
2	(2, T)	(1, F, F)	(F)
3	(3, F)	(2, T, F)	(F)
4	(4, F)	(3, T, T)	(T)

Chapter 3

Observability-Based Test Generation

In this chapter, we will illustrate the common issue impacting the efficacy of test suites generated to satisfy structural coverage criteria—*masking*—and formally define our solution—*observability* [65]. We then will describe how extending common structural coverage criteria to require observability can overcome masking, and consequently, sensitivity to program structure and oracle. Finally, we will describe how we implemented our tool to generate test obligations for observability-based coverage criteria.

3.1 Masking

Previous research has shown that the efficacy of test suites satisfying structural coverage criteria—defined over specific program elements such as control-flow branches, conditions, or decisions—can be highly sensitive to how expressions are written [23, 25, 50] and the selection of variables monitored by the test oracle [24, 59, 61]. This is due to *masking*, when the value of a program element—a variable or subexpression—is prevented from influencing the outcome of another expression.

In this work, we are primarily concerned with masking in terms of Boolean expressions. Masking occurs when the value of a condition (an atomic variable or subexpression) in a Boolean decision hides the effects of other conditions. We state a condition is masked if the outcome of a Boolean decision cannot by changed by varying the value of the condition while holding the rest of the conditions fixed (i.e., no matter what value the condition is, the final outcome of syntactic element of in-

terest does not change). For example, input `a = true` masks `b` in the decision `(a or b)`. As the decision's outcome is always `true`, regardless of the value of `b`, `a` is masked. Similarly, `a = false` masks `b` in decision `(a and b)`, as the decision will always evaluate to `false`.

By requiring that each condition demonstrate an independent influence on its decision's outcome, MC/DC is designed to prevent masking *within* an expression. Test cases must exist where, if we flip the value of a single condition while the others are held constant, the outcome of that decision must be changed. Branch, Decision, and Condition Coverage lack any such guarantee. This is one reason MC/DC is often required for testing of avionics and other safety critical systems—its requirements are more strenuous, but the additional assurances of the independent impact requirement theoretically increase the probability that logic faults will be detected.

```
1. v1 = in1 and in2;
2. out = v1 or in3;
```

Figure 3.1: Non-inlined Sample Code for MC/DC Criterion

```
1. out = ((in1 and in2) or in3);
```

Figure 3.2: Inlined Sample Code for MC/DC Criterion

However, how the code is structured has a major impact on the formulation of the test obligations for a criterion and the efficacy of the suites satisfying such obligations. Consider the code fragments in Figures 3.1 and 3.2. The two code fragments are *semantically* identical—they offer the same outcome—but are written in two different styles. The fragment in Figure 3.1 is split over two separate, simple equations (a *non-inlined* style). The fragment in Figure 3.2 is *inlined*—written as a single, complex expression.

As the obligations for criteria such as MC/DC are posed over individual program elements, the MC/DC obligations for the non-inlined version will be much simpler—and much more trivially satisfied—than the obligations for the inlined version. In the non-inlined version, for example, `in1` must be shown to overcome any masking from the value of `in2`. However, in the inlined version, `in1` must overcome masking from both `in2` and `in3`. As a result, MC/DC is much harder to satisfy over inlined implementations, and requires a larger number of test cases. However, the produced test suites tend to be far more effective [23, 50]. Therefore, we can see that traditional coverage criteria are sensitive to program structure.

Table 3.1: Sample Test Suites Satisfying MC/DC Criterion

TestSuite1 = {(F, T, T), (T, T, T), (T, F, F), (F, F, F)}
TestSuite2 = {(F, T, F), (T, T, F), (T, F, F), (F, F, T)}

Further, just because a condition is shown to influence the outcome of the decision it resides within, there is no assurance that the condition will influence the *program output*. Consider again the sample code fragment in Figure 3.1. For this code, we can create two sample test suites for (`in1`, `in2`, `in3`) that satisfy the MC/DC criterion. The test suites are listed in Table 3.1. In the first two test cases in TestSuite1, the influence of `in1` and `in2` are masked out by `in3` = `T` in the second statement. As a result, a fault that corrupts either condition would not be observed by monitoring the value of *out*. Unless our test oracle also monitors internal variable `v1`, the fault is masked. However, with `in3` = `F` in the first two cases in TestSuite2, the values of `in1` and `in2` in the first statement can have an impact on the final outcome of the program.

Because obligations are posed over individual program elements, and make no demands on what happens after that element is executed, masking can prevent triggered faults from being observed. Masking can be addressed through selection of the

correct oracle. For instance, by monitoring all internal state variables as well as all the outputs, making between statements is not an issue [60, 61, 24]. For example, in the case of Figure 3.1, if we monitor the value of *v1* during testing, failures introduced by `in1` or `in2` can be detected without changing test suites. However, monitoring and specifying expected values for all variables is generally prohibitively expensive (or outright infeasible). A subset of variables could be used, if carefully chosen, but this selection is also non-trivial to make.

An alternative approach is to strength the coverage criteria with conditions on execution along the path from the program element of interest to the output (or other chosen oracle variables). Such path conditions can ensure the *observability* of such elements when we test.

3.2 Observability

The observability of a program is the degree to which it is possible to infer the internal state of a system given the information that we can monitor from the program—generally through program output [61]. We say an expression in a program is *observable* in a test case if we can change only its value, keeping the rest fixed, and see the influence of this change in the result of the test case. Otherwise, if this update has no visible influence, we say the expression is *not observable* in that test case.

In theory, masking can be overcome by requiring observability from a test suite—in addition to the existing test obligations of a host coverage criterion. Informally, we can obtain observability of test obligations by requiring that the variable whose assignment contains a particular element of interest remains unmasked through a path to a variable monitored by the test oracle.

Although this notion of observability was previously defined as an explicit extension to MC/DC [65], such requirements can be imposed on any existing criterion over Boolean expressions. The path conditions of observability establish a masking-clear

path from an expression containing a program element of interest—one with obligations defined over it—to a monitored variable. In this study, we apply observability to Branch, Condition, Decision, and MC/DC Coverage.

To formally define how observability is established, we can view a deterministic program P containing expression e as a transformer from inputs I to outputs O : $P : I \rightarrow O$. We write $P[v/e_n]$ for program P where the computed value for the n^{th} instance of expression e is replaced by value v. Note that this is not a substitution. Rather, we replace a single instance of expression e rather than all instances, which is more akin to mutation. We state e is observable in test t if $\exists v.P(t) \neq P[v/e_n](t)$. This idea can be straightforwardly lifted from test cases to test suites.

This formulation is a generalization of the semantic idea behind masking MC/DC criterion [7], lifted from decisions to programs. In masking MC/DC, the main obligation is that, for each condition c in given decision D, there are a pair of test cases t_i and t_j ensuring that c is observable in D's outcome for both outcomes (`true` and `false`): $((D(t_i) \neq D[true/c_n](t_i)) \wedge ((D(t_j) \neq D[false/c_n](t_j)))$.

As implemented then, an observable path is one where the value of the expression of interest can influence the output. As we are concerned with obligations established over Boolean expressions, observability requires that the outcome of each assigned Boolean variable influence program output:

$$\begin{array}{l}(\forall b_n \in Bool(P)). \\ \quad ((\exists t \in T.\ (P(t) \neq P[true/b_n](t))) \wedge (\exists t \in T.\ (P(t) \neq P[false/b_n](t))))\end{array} \tag{3.1}$$

where T is a given test suite and $Bool(P)$ is the collection of all variables corresponding to Boolean expressions within the program for which we establish test obligations from the host criterion. For Decision, Condition, and MC/DC Coverage, this is all decisions in the program. For Branch Coverage, this is the subset of control-altering decisions.

Note that, under this definition, observability eliminates masking from the point where the expression *as a whole* is assigned a value to the program output. Within that expression, masking can still occur as impacted by the choice of host criterion. For instance, the obligations of MC/DC require that masking be overcome within that expression, but the obligations of Condition Coverage do not.

3.2.1 Tagged Semantics

The semantic definition for observability, defined above, is unwieldy for test generation and test measurement. The analysis would require two versions of the program running in parallel to check that the results match. Then, for test measurement, the test suite must be executed separately for *each pair* of modified programs.

In order to define an observability constraint that efficiently supports monitoring and test generation, we can approximate semantic observability using a tagged semantics approach [13]. Although we have separated the concept of observability from MC/DC to an addition to any host criterion, the tagged semantics can still be used directly with any existing observability-based criterion.

Each variable corresponding to a Boolean expression or atomic value in the program is assigned a tag, the observability of which is tracked through the execution of the a program. If a tag is propagated to the output—or any "monitored" internal variable—the corresponding path condition is considered to be fulfilled. More precisely, we track pairings of tag and concrete outcome. If a tagged variable appears more than once in a decision, a tag is assigned to each occurrence uniquely. We then examine the number of all possible pairs that have reached as output in some test in order to evaluate the coverage level for a test suite.

Formal tagging semantics have been defined for a set of expressions, an imperative command language, as well as a simple dataflow language (shown in Table 3.4). A reduction semantics with evaluation contexts (RSEC) [14] is used for presentation,

and the K tool suite [53] is used to check for consistency. The rules, which run over *configurations* containing K (the syntax being evaluated) and a set of configuration parameters being labeled, operate by applying rewrites at positions in syntax where the evaluation context allow. A *context* can be a program or program fragment with a *hole* (represented by $\Box$)—a placeholder where a rewrite can occur. In their definition, maps are assumed to have operations—lookup $(\sigma\, x)$ and update $\sigma[x \leftarrow \nu]$, the empty map $\emptyset$, and lists with concatenation $x.y$ and cons $elem :: x$, and operators. Additional syntax, which will be formatted as gray background to distinguish from user-level syntax, may be introduced during rewriting.

Table 3.2: Expression syntax, context, and semantics

E	::=	$Val \mid Id \mid E\ op\ E \mid not\ E \mid$
		$E\ ?\ E : E \mid tag(E,\ T) \mid$ (Val, TS) $\mid$ addTags(E, TS)
$Context$	::=	$\Box \mid Context\ op\ E \mid E\ op\ Context \mid not\ Context \mid$
		$Context\ ?\ E : E \mid addTags(Context,\ TS) \mid$
		$\langle \kappa : Context,\ \epsilon : Env, ... \rangle$
lit		$n \Rightarrow (n,\ \varnothing)$
var		$\langle \epsilon : \sigma \rangle\, [x] \Rightarrow\ \langle \epsilon : \sigma \rangle\, [(\sigma x)] \qquad if\ x \in dom(\sigma)$
op		$(n_0,\ l_0) \oplus (n_1,\ l_1) \Rightarrow\ (n_0\ \oplus\ n_1,\ l_0 \cup l_1)$
and$_1$		$(tt,\ l_0)\ and\ (tt,\ l_1) \Rightarrow\ (tt,\ l_0 \cup\ l_1)$
and$_2$		$(tt,\ l_0)\ and\ (ff,\ l_1) \Rightarrow\ (ff,\ l_1)$
and$_3$		$(ff,\ l_0)\ and\ \_ \Rightarrow\ (ff,\ l_0)$
or$_1$		$(ff,\ l_0)\ and\ (ff,\ l_1) \Rightarrow\ (ff,\ l_0 \cup\ l_1)$
or$_2$		$(ff,\ l_0)\ and\ (tt,\ l_1) \Rightarrow\ (tt,\ l_1)$
or$_3$		$(tt,\ l_0)\ and\ \_ \Rightarrow\ (tt,\ l_0)$
ite$_1$		$(tt,\ l_0)\ ?\ e_t\ :\ e_e \Rightarrow\ addTags(e_t,\ l_0)$
ite$_2$		$(ff,\ l_0)\ ?\ e_t\ :\ e_e \Rightarrow\ addTags(e_e,\ l_0)$
tag		$tag(t,\ (v,\ l)) \Rightarrow\ (v,\ l\ \cup\ \{(t,\ v)\})$
adt		$addTags((v,\ l_0),\ l_1) \Rightarrow\ (v,\ l_0\ \cup\ l_1)$

In Tables 3.2, 3.3, and 3.4, expressions yield $(Val,\ TS)$ pairs, where TS is a set of tags, and are evaluated in a context containing environment ε of type $Env\ =\ (id \rightarrow (Val\ x\ TS))$. The expressions are standard, except the $tag(E,\ T)$ which adds a tag to the set of tags associated with the expression e. For any structural coverage,

Table 3.3: Imperative command syntax, context, and semantics

S	::=	$skip \mid S;\ S \mid if\ E\ then\ S\ else\ S \mid$
		$Id\ :=\ E \mid while\ E\ do\ S \mid$ end(**List** Id, TS)
$Context$	::=	$\cdots \mid Id\ :=\ Context \mid if\ Context\ then\ S\ else\ S \mid$
		$Context;\ S \mid \langle \kappa : Context,\ \epsilon : Env,\ C : TS \rangle$
	asgn	$\langle \epsilon : \sigma > [x\ :=\ (n,\ l)] \Rightarrow < \epsilon : \sigma\,[x \leftarrow (n,\ l)] \rangle\,[skip]$
	seq	$skip;\ s_2 \Rightarrow s_2$
	cond$_1$	$\langle C : c \rangle\,[if\,(tt,\ l)\ then\ s_1\ else\ s_2] \Rightarrow$
		$\langle C : c\ \cup\ l \rangle\,[s_2;\ end\,(V,\ c)]$
		$where\ V = (Assigned\ s_1)\,.\,(Assigned\ s_2)$
	cond$_2$	$\langle C : c \rangle\,[if\,(ff,\ l)\ then\ s_1\ else\ s_2] \Rightarrow$
		$\langle C : c\ \cup\ l \rangle\,[s_1;\ end\,(V,\ c)]$
		$where\ V = (Assigned\ s_1)\,.\,(Assigned\ s_2)$
	while	$while\,(e)\ \ s \Rightarrow if\,(e)\ \ then\,(s;\ \ while\,(e)\ \ s)\ \ else\ skip$
	endcond$_1$	$\langle C : c' \rangle\,[end\,(nil,\ c)] \Rightarrow \langle C : c \rangle\,[skip]$
	endcond$_2$	$\langle \epsilon : \sigma,\ C : c' \rangle\,[end\ \ (x\ ::\ V,\ c)] \Rightarrow \langle \epsilon : \sigma',\ C : c' \rangle\,[end\,(V,\ c)]$
		$where\,(\sigma\ x)\ =\ (n,\ l)\ \ and\ \sigma'\ =\ \sigma\,[x \Leftarrow (n,\ l \cup\ c')]$
	prog	$s \Rightarrow \langle \kappa :\ s,\ \epsilon :\ \varnothing,\ C\ :\ \varnothing \rangle$

it is assumed that each Boolean variable is wrapped in a *tag* expression. Masking is defined by operators: 1) *and*—given (*a and b*), for *a* is not masked out, *b* has to be *true*, so the tag assigned to *a* propagates only when *b* is *true* (and vice-versa); 2) *or*—given (*a or b*), for *a* is not masked out, *b* has to be *false*, so the tag assigned to *a* propagates only if *b* is *false* (and vice-versa); 3) *ite*—given (*if a then b else c*), for *b* is not masked out, *a* must be *true*, therefore *b*'s tag propagates when *a* is *true*; similarly, *c*'s tag propagates when *a* is *false*; 4) relation expressions such as $a > b$, *a* and *b* are never masked out by each other; these will not be shown in table 3.2.

The imperative language semantics define the way tags broadcast through commands: tags need to propagate through all variables assigned in either branch in conditional statements, for the value of a variable can be influenced by not being assigned by the condition. $C\ :\ TS$ is introduced into the expression configuration to store the set of variable tags. Once a statement has been executed, the tags added

Table 3.4: Dataflow program syntax, context, and semantics

EQ	::=	$Id \ = \ E \mid id \ = \ pre\,(E)$
$Prog$	::=	$(I,\ Env,\ \textbf{List}\ EQ)$
$Context$	::=	$\cdots \mid Context;\ \textbf{List}\ EQ \mid Context \ ::\ \textbf{List}\ EQ \mid$
		$EQ \ ::\ Context \mid Id \ = \ Context \mid Id \ = \ pre(Context) \mid$
		$\langle \kappa : Context,\ \tau :\ \textbf{List}\ Env,\ O : \textbf{List}\ Env,\ \epsilon \ :\ Env,\ S \ :\ Env \rangle$
comb		$\langle \epsilon : \sigma \rangle\ eqs_0.\,((x \ = \ (n,\ l)) \ ::\ eqs_1) \Rightarrow$
		$\langle \epsilon :\ \sigma\,[x \leftarrow \ (n,\ l)] \rangle\ eqs_0.eqs_1$
state		$\langle S : \sigma \rangle\ eqs_0.\,((x \ = \ (n,\ l)) \ ::\ eqs_1) \Rightarrow$
		$\langle S : \sigma\,[x \Leftarrow (n,\ l)] \rangle\ eqs_0.eqs_1$
write		$\langle O : \kappa,\ \epsilon : c \rangle\ nil;\ eqs \Rightarrow \langle O : \kappa.\,[c]\,,\ \epsilon : c \rangle\ eqs$
cycle		$\langle \tau : \sigma_i \ ::l,\ \epsilon :\ \_,\ S : \sigma_l \rangle\ eqs \Rightarrow$
		$\langle \tau : i,\ \epsilon : (\sigma_i \ \cup \ \sigma_l)\,,\ S :\ \varnothing \rangle\ eqs;\ eqs$
prog		$(i,\ s,\ eqs) \Rightarrow \langle \tau :\ i,\ O :\ nil,\ S : s,\ \epsilon : \varnothing,\ \kappa :\ eqs \rangle$

to C by conditional statements will be removed. An *end* statement is introduced to implement that—it is appended to clear C and propagate the conditional tags to all variables assigned in the conditional body. A helper function ($Assigned\ s$) will then return the list of variables assigned in s. Given a program (or program fragment) containing inputs, the rules defined in table 3.3 will determine the set of tags propagating to output.

Dataflow languages, such as Simulink and SCADE, are popular for model-based development, and assign values to a set of equations in response to periodic inputs. To store system state, state variables ($\frac{1}{z}$ blocks in Simulink) are used. Our dataflow language consists of assignments to combinatorial and state variables, and the semantics are defined over lists (traces) of input variable values. The expression configuration is extended to contain an input trace I, output trace O, and state environments S. Evaluation proceeds by cycles: at the beginning of a cycle, the *cycle* rule constructs the initial evaluation environment.

During a cycle, variable values are recorded using the *comb* and *state* rules. Note that the context does not force an ordering on evaluation of equations; instead, an

equation can evaluate as soon as all variables it uses have been stored in the environment. When all equations have been computed, the *write* rule appends the environment to the output list. The *prog* rule, given an input list, an initial state environment, and a list of equations, initializes the configuration for the *cycle* rule. Coverage can be determined by examining the tags stored in the output environment list.

Note that both the tagging semantics are *optimistically inaccurate* with respect to observability; that is, they may report that certain conditions are observable when they are not. This is easily demonstrated by a small code fragment:

```
if (c) then out := 0 else out := 0 ;
```

The semantic model of observability will correctly report that `c` is not observable; it cannot affect the outcome of this code fragment. However, the tagging model propagates the tags of `c` to the assignments in the `then` and `else` branches.

3.3 Model-Based Test Generation

In model-based test generation, models are annotated with *trap properties*. A property of interest is negated, then the model checker returns a counterexample—a test input sequence demonstrating that the property can be met. In order to generate tests that meet the conditions of observability, we need to be able to annotate the program with trap properties that track the tags described above. This is accomplished by conjoining the coverage obligations of the host criterion with a path condition representing the variable in which the test obligation's target resides. Observability can be attained either *immediately*—within the current computational cycle—or *after a delay.* Path conditions must reflect either case. In this section, we describe this annotation for the Lustre dataflow language [29].

3.3.1 Immediate Non-Masking Paths

A variable `x` is observable if a computational path can be found from `x` to a monitored variable `z` in which `x is not masked.` If such a path can be taken entirely within one computational step, we call it a *immediate non-masking path*, and variable `x` is *immediately observable*. Such paths can be defined inductively by examining the variables that use `x` in their definition. For example, if `x` is used in the definition of variable `y`, and `x` is not masked by other variables within that definition, then `x` is immediately observable at `y`. We can then consider the variables that use `y` in their definition, and apply the same criteria.

We track such notions by introducing additional variables. First, *combinatorial usage expressions*—`x_COMB_USED_BY_y`—determine whether a variable is masked within a definition. The variable is `true` if `x` is not masked by other elements of `y`'s definition. Second, *immediate observability expressions*—`x_COMB_OBSERVED`—which offer a way to check the status of the non-masking path. For each Boolean variable in the program, there could exist one or more immediate non-masking paths.

Consider the code fragment in Figure 3.3, where `out` is an output variable, `in1` and `in2` are input variables, and `v1`, `v2`, and `v3` are internal variables.

```
v1 = in1 and in2;
v2 = if (in3) then v3 else v1;
v3 = not in2;
out = v1 or v2;
```

Figure 3.3: Sample Lustre code

From the equations, we can generate additional definitions to track the observability of variables as in Figure 3.4. Variable `v1` is used by two variables—`v2` and `out`—in their definitions and therefore has two potential immediate non-masking paths: directly through the output variable `out` or through `v2`. Variable `in2` also has two

potential immediate non-masking paths: either through `v1` or through `v3`. All the other variables are each used once, so each has only one immediate non-masking path.

```
in1_COMB_USED_BY_v1 = in2;
in2_COMB_USED_BY_v1 = in1;
in3_COMB_USED_BY_v2 = true;
v3_COMB_USED_BY_v2 = in3;
v1_COMB_USED_BY_v2 = (not in3);
in2_COMB_USED_BY_v3 = true;
v1_COMB_USED_BY_out = (not v2);
v2_COMB_USED_BY_out = (not v1);

out_COMB_OBSERVED = true;
in1_COMB_OBSERVED = (in1_COMB_USED_BY_v1 and v1_COMB_OBSERVED);
in2_COMB_OBSERVED = ((in2_COMB_USED_BY_v1 and v1_COMB_OBSERVED) or
    (in2_COMB_USED_BY_v3 and v3_COMB_OBSERVED));
in3_COMB_OBSERVED = (in3_COMB_USED_BY_v2 and v2_COMB_OBSERVED);
v3_COMB_OBSERVED = (v3_COMB_USED_BY_v2 and v2_COMB_OBSERVED);
v1_COMB_OBSERVED = ((v1_COMB_USED_BY_v2 and v2_COMB_OBSERVED) or
    (v1_COMB_USED_BY_out and out_COMB_OBSERVED));
v2_COMB_OBSERVED = (v2_COMB_USED_BY_out and out_COMB_OBSERVED);
```

Figure 3.4: Introduced variables to track immediate non-masking paths

3.3.2 Delayed Non-Masking Paths

Reactive systems compute in cycles, and variable values from the previous cycle can be referred to. As a result, the effect of a variable on output may not be observed until several computation cycles after a value is computed. In each of these intermediate computational steps, the system state is stored in a delay variable, until it propagates to an output eventually. We call such a path—propagating influence through a delay variable to an output—a *delayed non-masking path* and the variable is *delay observable*. A delayed non-masking path can be built over multiple immediate non-masking paths: from a variable to a latch—a delay variable—then from the latch to another latch, and so on, until an output is reached.

Suppose we have a sample code fragment in Figure 3.5, where `delay1` and `delay2` are delay expressions.

```
delay1 = (0 -> pre(in1));
v1 = (if (delay1 > 0) then true else in2);
delay2 = (false -> pre(v1));
```

Figure 3.5: Sample Lustre code

As with immediate non-masking paths, we can inductively build paths involving delay expressions. An example can be seen in Figure 3.6. Variable `v1`, which uses `delay1` and `in2` in its definition, is a used in the definition of delay expression `delay2`. Therefore, a delayed non-masking path from `delay1` to `delay2` is composed of the immediate non-masking path from `delay1` to `v1`, then a delayed non-masking path from `v1` to `delay2`.

```
delay1_COMB_USED_BY_v1 = true;
in2_COMB_USED_BY_v1 = (not (delay1 > 0));

in1_SEQ_USED_BY_delay1 = true;
v1_SEQ_USED_BY_delay2 = true;
delay1_SEQ_USED_BY_delay2 = (delay1_COMB_USED_BY_v1 and
    v1_SEQ_USED_BY_delay2);
in2_SEQ_USED_BY_delay2 = (in2_COMB_USED_BY_v1 and
    v1_SEQ_USED_BY_delay2);
```

Figure 3.6: Introduced variables to track delayed non-masking paths

This annotation gives us the means to track immediate paths to latches. However, it is still necessary to establish the means to knit these paths together to form the sequential path over one or more delays passed on the path to output. To do so, we introduce a *token* mechanism—a special variable to mark the current delay location. Once the token is initialized to a delay variable `x`, it can non-deterministically move to any other delay location—as long as `x` can be sequentially used by that location.

It can also move to a special TOKEN_OUTPUT_STATE, is a monitored variable is reached or TOKEN_ERROR_STATE is the token can no longer possible be observed through a monitored variable or another delay.

We generate token equations to track the path taken through delay variables. Consider the code fragment in Figure 3.7.

```
v1 = (false -> (not (pre v2)));
v2 = (false -> (pre v1));
v3 = (0 -> (if ((pre v3) = 3)
           then 0
           else ((pre v3) + 1)));
out = (((v1 and v2) and (v3 = 2)) or
       ((not (v1 and v2)) and (not (v3 = 2))));
```

Figure 3.7: Sample Lustre code

We can generate the token equations shown in Figure 3.8. In this case, if we are currently at `TOKEN_D1`, and `v1` is immediately observable, then we reach the output. Otherwise, if `v1` can be delay observed through `v2`, then the token moves to `TOKEN_D3`.

```
token_next = (if ((pre token) = TOKEN_INIT_STATE) then token_first
    else (if ((pre token) = TOKEN_ERROR_STATE) then TOKEN_ERROR_STATE
    else (if ((pre token) = TOKEN_OUTPUT_STATE) then TOKEN_OUTPUT_STATE
    else (if ((pre token) = TOKEN_D1) then
      (if v1_COMB_OBSERVED then TOKEN_OUTPUT_STATE
      else (if ((token_nondet = TOKEN_D3) and v1_SEQ_USED_BY_v2)
      then TOKEN_D3 else TOKEN_ERROR_STATE))
    else (if ((pre token) = TOKEN_D2) then
      (if v3_COMB_OBSERVED then TOKEN_OUTPUT_STATE
      else (if ((token_nondet = TOKEN_D2) and v3_SEQ_USED_BY_v3)
      then TOKEN_D2 else TOKEN_ERROR_STATE))
    else (if ((pre token) = TOKEN_D3) then
      (if v2_COMB_OBSERVED then TOKEN_OUTPUT_STATE
      else (if ((token_nondet = TOKEN_D1) and v2_SEQ_USED_BY_v1)
      then TOKEN_D1 else TOKEN_ERROR_STATE))
    else TOKEN_ERROR_STATE))))));
```

Figure 3.8: Example token equations

3.3.3 Test Obligations

A test obligation serves as a partial test case specification which requires some property deemed important to thorough testing. Observability-based test obligations conjoin the base obligations fulfilling the host coverage criterion (e.g., MC/DC) with the path conditions required to establish either an immediate non-masking path or a delayed non-masking path from the expression where the base obligation is established to a monitored variable. An example obligation is shown in Figure 3.9.

```
v2_AT_v1_TRUE = in1 and delay2;
v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE = v2_AT_v1_TRUE and
    (v1_SEQ_USED_BY_delay1 and token=delay1);
v2_AT_v1_TRUE_CAPTURED = v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE ->
    (v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE or
    pre(v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE));

  obligation_0 = ((v2_AT_v1_TRUE and v1_COMB_OBSERVED) or
      (v2_AT_v1_TRUE_CAPTURED and token = TOKEN_OUTPUT_STATE));
```

Figure 3.9: Sample test obligations

Expression `v2_AT_v1_TRUE` is a base obligation from a host criteria, defining an MC/DC obligation in expression `v1`. For delayed non-masking paths, we have to define the instant in which the expression would be immediately observable at a delay (the moment of *capture*). We then must latch this fact for the remainder of execution, in case the execution path hits a monitored variable. Expressions `v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE` and `v2_AT_v1_TRUE_CAPTURED` define this concept of capture for delayed non-masking paths. Finally, the full obligation is defined in expression `obligation_0`. In the obligation, the subexpression before the `or` operator defines immediate observability, and the second subexpression defines delayed observability. If either path is observed, then the obligation is met.

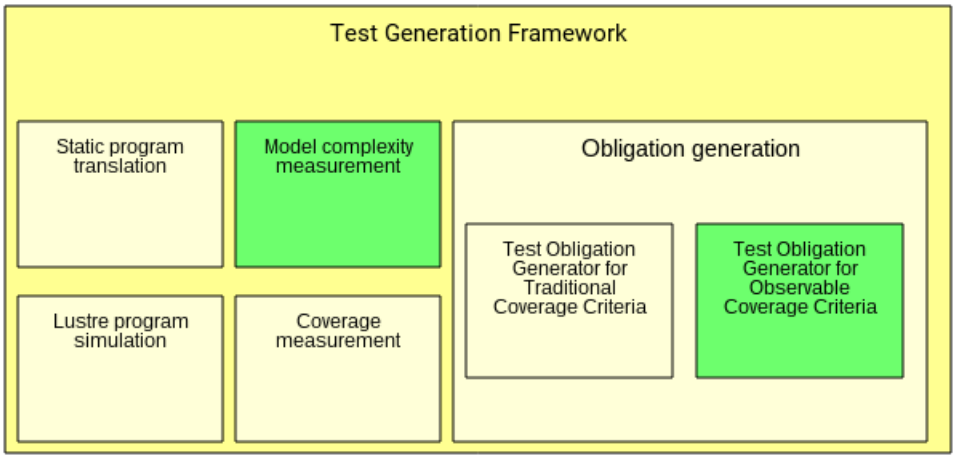

Figure 3.10: Components of Lustre project

3.4 Test Generation Framework

We choose Lustre, an open source project developed in Java language based on the JKind model checker, as the framework. Lustre provides a set of tools (four major features) to generate tests for Lustre programs: static program translation, coverage obligation generation (for traditional structural coverage criteria), coverage measurement, and Lustre program simulation. We implemented our work in this framework to provide test obligation generation for observability-based coverage criteria and model complexity measurement (main functional components of Lustre are shown in Figure 3.10). In this section, we will briefly describe the implementation of our work.

3.4.1 Observability-Based Coverage Obligation Generation

As we discussed in previous section, we generate test obligations for observable coverage criteria, by combining observability with original coverage obligations. To implement the feature, we introduced an agent—ObservabilityCoverage—to take over task regarding observable coverage obligation generation. It first gathered all path-constraint related information by calling corresponding functions then generated ob-

servable coverage obligations by requiring observable path constraints over the original obligations, and returned to LustreCoverage where the generated obligations, as part of the output program, would be written into a file.

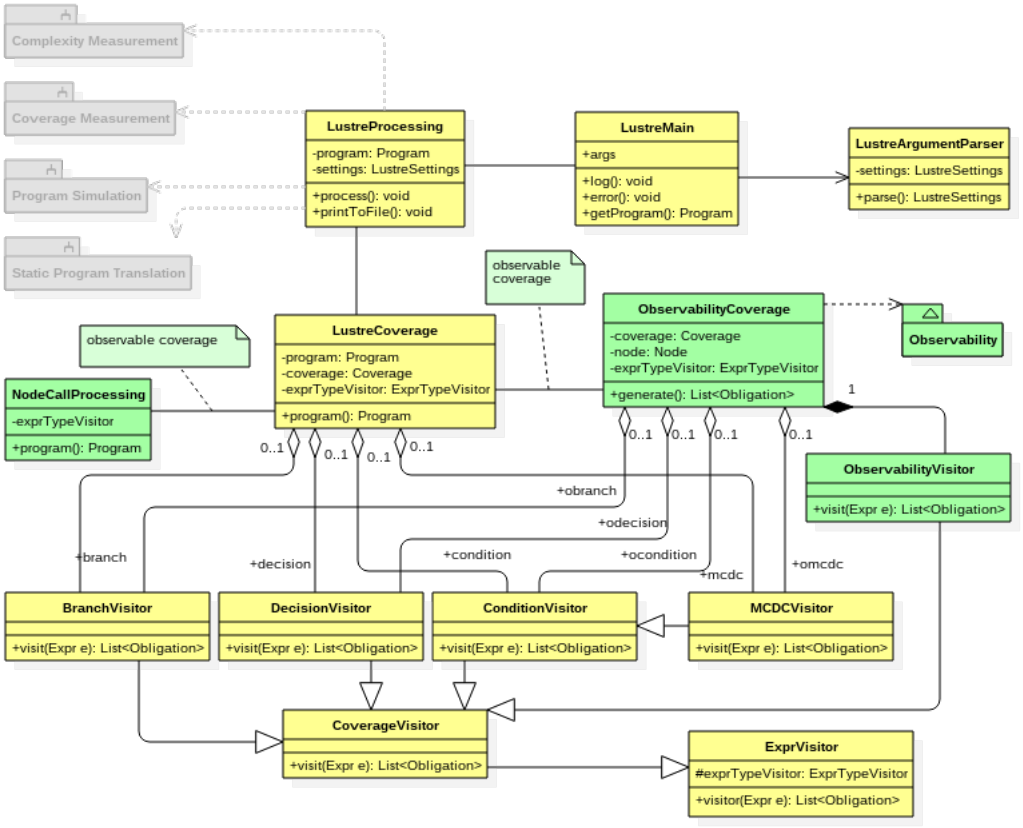

Figure 3.11: Simplified class diagram of coverage obligation generation model

Figure 3.12 presents the class diagram of the model of observability coverage obligations and the sequence diagram is shown in figure 3.13. When LustreCoverage detected an observability-based coverage, it dispatched the obligation generation job to ObservabilityCoverage. ObservabilityCoverage first collected the corresponding original obligation from related generation function, and called ObservabilityParser first to parse the program, node by node, crawling all the definition-dependent relations between variables, from an output or a latch (i.e., delay-containing expression). Tracks starting from an output form a combinatorial observable tree (referred as

combUsedTree) for the output, a delay observable tree (referred as *seqUsedTree*) for a latch. After collected all the observable trees, ObservabilityCoverage then passed them to some functions to build combinatorial non-masking paths and delayed non-masking paths. At last, ObservableCoverage called ObservableObligation to generate observable obligations by combining the original coverage obligations and the non-masking path constraints.

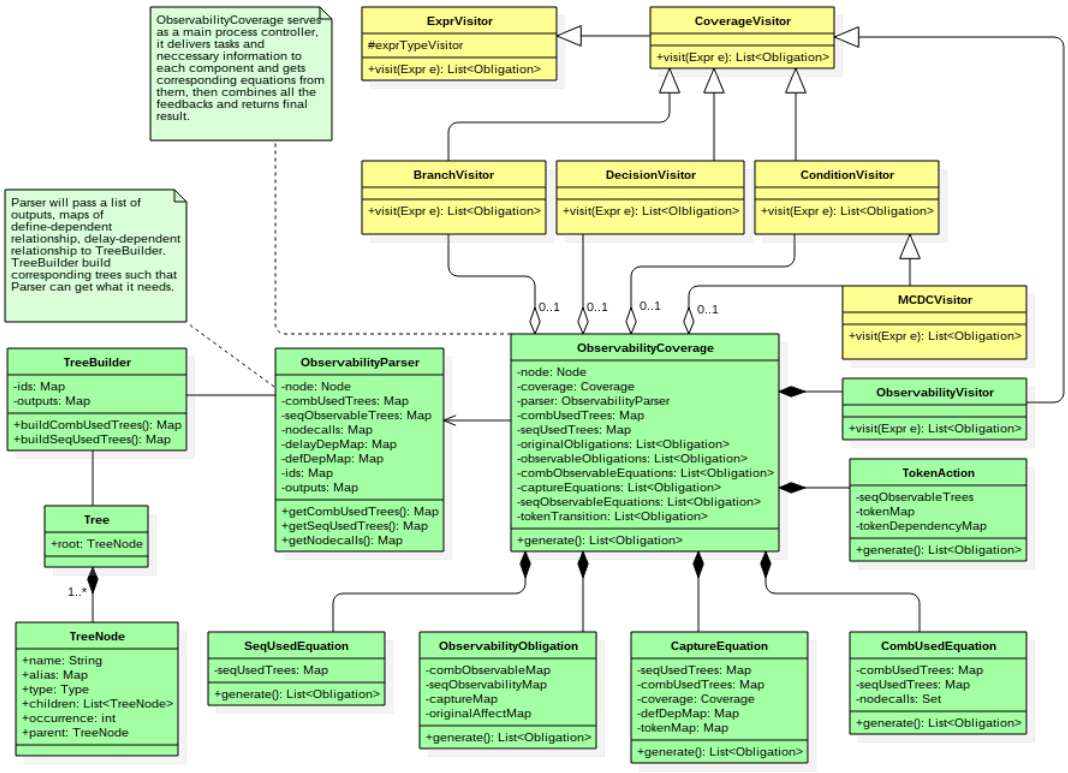

Figure 3.12: Simplified class diagram of observability coverage generation model

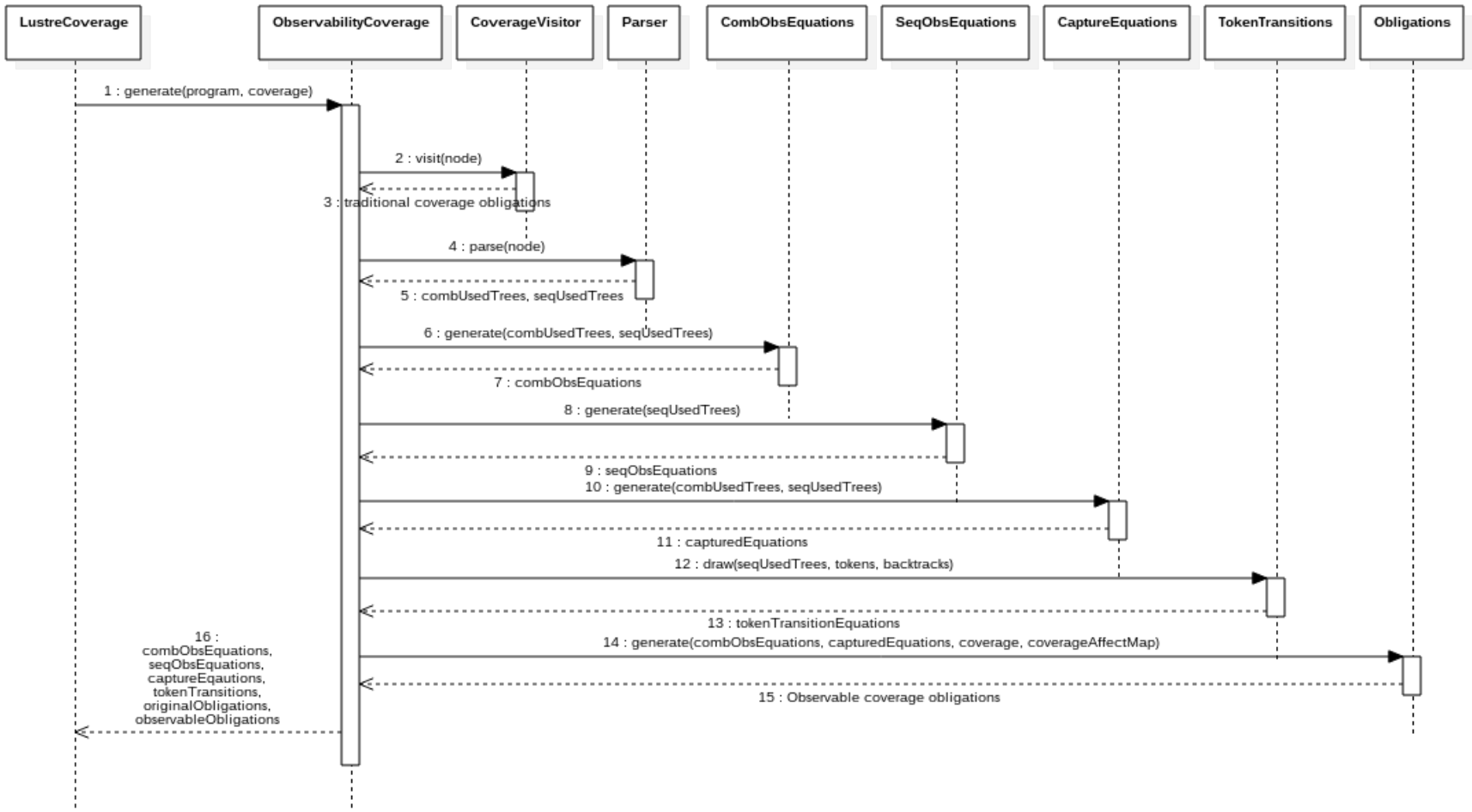

Figure 3.13: Simplified sequence diagram of observability coverage generation model

PARSE

We first parsed given node, extracting the definition-dependent relations between variables from each output to inputs or latches, then formed an *immediately combinatorial observable* tree (i.e., *combUsedTree*) for one output. The core algorithm of building a *combUsedTree* is presented in Algorithm 1 and 2. Each *combUsedTree* is rooted by an output and for each node in such tree, all the children are used in the node's definition. We stopped building such tree when an input or a latch (delay-containing equation) was met. Therefore, a bottom-up path, from any node in the path to the root, in a *combUsedTree* represents the *combinatorial non-masking* path, from the certain variable to an output.

Algorithm 1 Build combinatorially observable trees, one tree per output variable.

```
1: function combUsedTree
2:     for each output ∈ outputs do
3:         root ← TreeNode(output)
4:         buildTree(root)
5:         trees ← ⟨output, root⟩
6:     end for
7:     return trees
8: end function
```

Algorithm 2 Given a root node, build a parse tree to record all the definition-dependent relations from the root to inputs or a delay expression.

```
1: procedure buildTree(root)
2:     queue ← root
3:     while queue ≠ ∅ do
4:         node ← queue.head
5:         node.children ← var ∈ def(node)   ▷ var is used in the definition of node
6:         queue ← child ∈ node.children
7:     end while
8: end procedure
```

A delay observable tree (i.e., *seqUsedTree*) tracked the *delayed non-masking* paths from inputs or latches to a latch. A *seqUsedTree* is rooted by a latch, it first records the immediately observations between the latch and delay variables used in its def-

Algorithm 3 Build delay observable trees from each delay state to inputs and non-directly observable delay variables.

```
1: function seqUsedTree
2:     seqRoots ← id with a delay-containing definition
3:     for each seqRoot ∈ seqRoots do
4:         root ← TreeNode (seqRoot)
5:         root.children ← var ∈ def (seqRoot)        ▷ var is used in the definition of seqRoot
6:         for each subroot ∈ root.children do
7:             buildTree (subroot)
8:         end for
9:         trees ← ⟨seqRoot, root⟩
10:    end for
11:    return trees
12: end function
```

inition (the first two levels in the tree), then the immediately observations between variables (from nodes in the second level to leaf nodes) which is the same logic in a *combUsedTree*. The core algorithm of building a *seqUsedTree* is shown in Algorithm 3 (the first two levels) and Algorithm 2 (the rest levels). Therefore, a bottom-up path in a *seqUsedTree*, from any node to the root, represents the path the variable is immediately observable at some delay variable (i.e., the children of the root latch) which is sequentially used by the root latch.

Draw Token Transitions

Token transition is the mechanism to knit all the *non-masking* paths to to form a *delayed non-masking* path through one or more delays to an output. Algorithm 4 shows the core algorithm of generating the state automaton of tokens. For a latch (i.e., the current token state, denoted as *TOKEN_D1* and the like in tagged semantics), we crawl the program for all the latches or some output that used it in their definitions, each of which is the next token state (for another latch) or output state (for an output). That is, under certain condition—the latch or output uses current latch in its definition, and the computational or delayed path to the latch or output is not

masked out—the affect of some variable propagated to current delay state can be passed to next delay state (if some delay observable path found) or directly to output state (if some immediately observable path found), or an error state (if no observable path found or some error occurred).

Algorithm 4 Draw token transition.

```
1: function tokenTrans
2:     trans ← ∅
3:     for each token ∈ TOKEN do
4:         for each next ∈ {token' | token' ∈ TOKEN & token' ∈ DEPEND(token)} do
5:                 ▷ token' is a token AND token is used in the definition of token'
6:             trans ← ITE(token_next = next & next.seqUsed(token), next, trans)
7:         end for
8:         trans ← ITE(combObservable(token), OUT_STATE , trans)
9:         trans ← ITE(pre_token = token, trans, ERR_STATE)
10:    end for
11:    return trans
12: end function
```

Generate Observable Coverage Obligations

Algorithm 5 Add tags for original coverage obligations.

```
1: function addTag(coverage)
2:     tags ← ∅
3:     for each obligation ∈ {o | o is an obligation satisfying the original coverage}
   do
4:         exprTag ← ∅
5:         for each component ∈ {e | e ∈ obligation.expr AND
                                    e is a concern of specifying coverage} do
6:                 ▷ tag, ⟨value of the component, tag assigned to the component⟩
7:             for each value ∈ {true, false} do
8:                 exprTag ← ⟨component.id, value⟩
9:             end for
10:        end for
11:                ▷ tag, ⟨equation, list of tags of component in the equation⟩
12:        tags ← ⟨obligation.id, exprTag⟩
13:    end for
14:    return tags
15: end function
```

From the original coverage obligations (an obligation is represented as an equation), We first extract the definition-dependent relations between each variable and the component that is concerned by the specifying coverage. The core algorithm is shown in Algorithm 5. First, for each component under concern in an equation, we track the pair of a tag assigned to the component and the value of the component. We then add a tag to the set of tags associated with the equation once we have labeled all concerned components in it. This process is repeated until all equations have been tagged.

Algorithm 6 Generate "capture" equations

```
1:  function CAPTUREEQUATION
2:      capture ← ∅
3:      for affect ∈ tags.KEYSET do
4:          for var ∈ tags.get(affect) do
5:              for i = 0 to var.occurrence do          ▷ each occurrence is distinct
6:                  for value ∈ {true, false} do
7:                      var.affectAt(value, affect)
8:                      for each latch ∈ LATCH AND
                                    affect is sequentially observable at latch do
9:                          1. build a sequential observable path (for specifying value)
                            from affect to each latch
10:                         2. concatenate all such paths with OR operation
11:                         3. assign the final result of step 2 to equation
12:                     end for
13:                     capture ← ⟨var.affect, equation⟩
14:                 end for
15:             end for
16:         end for
17:     end for
18:     return capture
19: end function
```

Usually, given a Lustre program, we can propagate the affect on a variable (i.e., internal state of the program) to more than one target states in a certain execution step and for a certain latch, it could have several source latches propagating affect from. So the delay observable equations are not enough, we need a means to determine where the affect was propagated from in a computational cycle.

As it is shown in the core algorithm (Algorithm 6), To generate these "capture" equations, we iterate all the tag sets we just extracted,

1. for each tag in a set, we backtracked from each tag—each tag is associated with a component which was concerned by the coverage, searching for latches that used the corresponding component in their definitions;

2. we then connected all immediate and delayed non-masking paths (with the corresponding token states) with an OR operation, which represents the multiple possible paths to propagate the affect to following step.

3. at last we use the result of step 2 to form the equation describing in current execution cycle through which path the influence was passed.

Algorithm 7 Generate observable coverage obligations.

```
1:  function generateObligation(coverage)
2:      obligations ← ∅
3:      for key ∈ affectAtCaptures do
4:          for var ∈ affectAtCaptures.get(key) do
5:              for i = 0 to var.occurrence do          ▷ each occurrence is distinct
6:                  var.notMasked(value, key, coverage)
7:                  1. build a non-masking path (for specifying value)
                        from var to key
8:                  2. concatenate the non-masking path with token transition
9:                  3. assign the final result of step 2 to obligation
10:                 obligations ← ⟨id, obligation⟩
11:             end for
12:         end for
13:     end for
14:     return obligations
15: end function
```

After we have collected the original coverage obligations, added all observable path constraints, and defined how values of concerned code component propagate through one or more delays to an output, we can start generating obligations for corresponding observable version. The core algorithm is described in Algorigthm 7.

3.4.2 Model Complexity Measurement

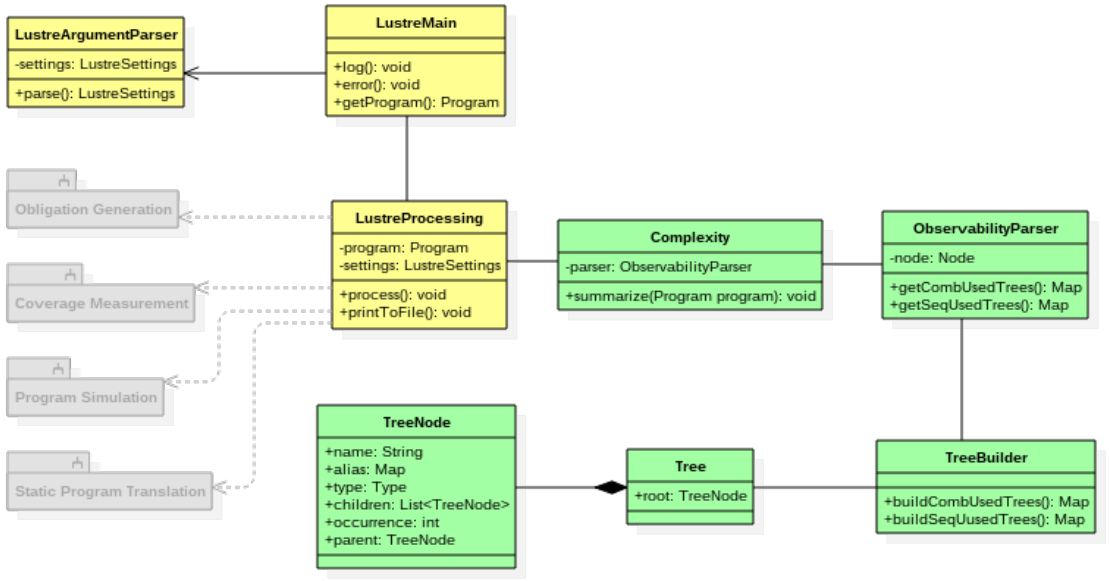

Figure 3.14: Simplified class diagram of Lustre model complexity measurement

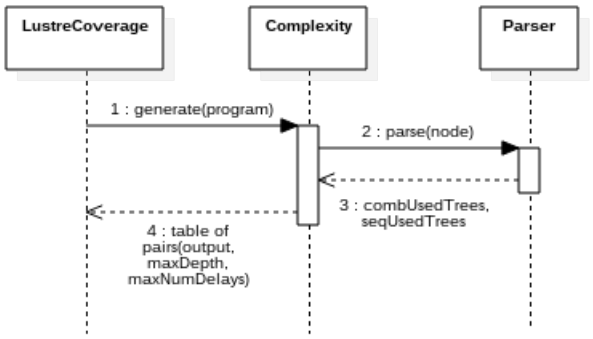

Figure 3.15: Simplified flowchart of Lustre model complexity measurement

Addition to observable coverage generation, we implemented a standalone function to measure the complexity—maximum depth of immediately observable paths and maximum number of immediately observable delay states—of given Lustre program. Rather than output the two maximum values, we output the depth and the

Algorithm 8 Complexity measurement—get the depth of immediately observable paths from each output

```
1: function HEIGHT
2:     if root = NULL then return 0
3:     end if
4:     queue ← root
5:     height ← 0
6:     while queue ≠ ∅ do
7:         height ← height + 1
8:         size ← queue.size
9:         for i = 0 to size do
10:            node ← queue.head
11:            queue ← node.children
12:        end for
13:    end while
14:    return height
15: end function
```

Algorithm 9 Complexity measurement—get the number of latches that is immediately observable to each output

```
1: function NUMOFLATCHES(root)
2:     count ← 0
3:     for each leaf ∈ root.leaves do
4:         if leaf ∈ SET (latchs) then
5:             count++
6:         end if
7:     end for
8:     return count
9: end function
```

number of immediately observable delays instead, to show more details of the model. We show the simplified class diagram and sequence diagram in Figure 3.14 and 3.15. Algorithm 8 presents the core algorithm of calculating the maximum depth of immediately non-masking paths starts from the chosen output. Algorithm 9 presents the core algorithm of counting the number of delay states that are immediately observable at a specifying output.

Chapter 4

Study

We wish to assess the quality—in terms of fault finding—of test suites generated to satisfy both observable and traditional versions of the studied coverage criteria. We also want to evaluate the effect of observability on the effectiveness of test suites. Thus, we address the following questions:

1. Which criterion has the highest average likelihood of fault detection?

2. Are test suites generated to satisfy observable variants of coverage criteria more effective than the test suites generated to satisfy the original criterion?

The first question allows us to establish a baseline for discussion, and a general ranking of criteria. Which criterion—whether observable or traditional—returns the best results, on average? In the second case, we wish to understand whether observability generally offers a *beneficial* effect—does it consistently improve the likelihood of fault detection?

Additionally, we are interested in the nature of the tests generated to satisfy observable and traditional coverage criteria, and the effect of adding observability constraints to a coverage criterion:

3. What impact does observability have on the average size of the generated test suites and the average percentage of satisfied obligations for each criterion?

4. Across the studied criteria, does observability have a consistent effect on efficacy in terms of factors such as likelihood of fault detection, oracle and structure

sensitivity, and satisfiability of obligations?

Question 3 allow us to examine how the addition of observability impacts suite size and the ability of the test case generation process to satisfy the imposed test obligations. Question 4 allows us to examine the *impact of the choice of criterion*. That is, does it matter whether we start with MC/DC or Branch Coverage? Does observability consistently impact generated test suites?

In order to answer these questions, we have performed the following experiment for two sets of case examples (see Section 4.1):

1. **Generated mutants:** We generated up to 500 mutants, each containing a single fault. (Section 4.2.)

2. **Generated structural tests:** We generated test suites intended to satisfy Branch, Condition, Decision, and MC/DC Coverage—as well as observable variants of each—using counterexample-based test generation. (Section 4.3.)

3. **Reduced test suites:** We generated 50 reduced test suites using the test data generated in the previous step. (Section 4.4.)

4. **Computed effectiveness:** We computed the fault finding effectiveness of each test suite using both an output-only oracle and an oracle considering all outputs and internal state variables (a *maximally powerful* oracle) against the set of mutants. (Section 4.5.)

4.1 Case Examples

In this study, we have made use of two pools of systems. The studied systems were originally modeled using the Simulink and Stateflow notations [42, 43]. Then, each was translated to the Lustre synchronous programming language [28] to take advantage of existing automation. In practice, Lustre would be automatically translated

to C code. This is a syntactic transformation, and if applied to C, the results of this study would be identical.

Note that Lustre systems, and the original Simulink and Stateflow systems from which they were translated, operate in a sequence of computational steps. In each step, input is received, internal computations are performed sequentially, and output is produced. Within a step, no iteration or recursion is done—each internal variable is defined, and the value for it computed, exactly once. The system itself operates as an large loop.

4.1.1 Rockwell Collins Dataset

Table 4.1: Rockwell (Non-inlined) Example Information

Model	# Inputs	# Internal Variables	# Outputs
DWM1	11	569	7
DWM2	31	115	9
Latctl_Batch	23	128	1
Microwave	13	162	4
Vertmax	40	30	2

Table 4.2: Rockwell (Inlined) Example Information

Model	# Inputs	# Internal Variables	# Outputs	Average Complexity
DWM1	11	21	7	95.89285714
DWM2	31	10	9	21.36842105
Latctl_Batch	23	19	1	5.714285714
Microwave	13	99	4	9.15
Vertmax	40	30	2	720.5

The first set of systems consists of four industrial systems developed by Rockwell Collins engineers. Two of these systems, *DWM_1* and *DWM_2*, represent portions of a Display Window Manager for a commercial cockpit display system. The other two systems—*Vertmax_Batch* and *Latctl_Batch*—represent the vertical and lateral mode logic for a Flight Guidance System (FGS). In addition, we have used a Microwave System—control software for a generic microwave oven developed as a non-proprietary teaching aid at Rockwell Collins. This set of benchmarks has been used in previous

model-based test generation research [23, 24, 25, 22, 62, 60, 61], including previous work studying Observable MC/DC [65].

Previous work has found that, due to masking, the structure of the model can have a significant impact on the resulting efficacy of generated test suites for MC/DC [23, 50]. In theory, observability can assist in overcoming masking. To study this, we have generated two variants of each of the Rockwell Collins systems:

- **Maximally Non-Inlined:** Each expression is as simple as it can possibly be, with sub-expressions split into independent intermediate variable calculations.
- **Maximally Inlined:** Each expression is as complex as it can possibly be, with no intermediate sub-expressions used.

We repeat the entire experiment with both variants, in order to more thoroughly study the interaction between program structure and observability.

Information related to the non-inlined version of each system is provided in Table 4.1, and information related to the inlined versions is provided in Table 4.2. In both cases, we list the number of input variables, number of internal variables, and number of output variables. The latter two numbers give an indication of the size of the model, as each internal and output variable corresponds to an expression that must be calculated each computational cycle. For the inlined versions, we also list the average *complexity* of the inlined expressions—that is, the average number of boolean operations in each expression for that model.

4.1.2 Benchmarks Dataset

While the Rockwell Collins systems allow us to take a detailed look at the effect of program structure, the number of systems is relatively low. In order to more thoroughly analyze the effects of observability, we have also chosen an additional 40

Table 4.3: Benchmark Example Information

Model	# Inputs	# Internal Variables	# Outputs	Average Complexity
6counter	1	4	1	3.5
AlarmFunctionalR2012	44	182	5	9.086666667
CarAll	2	8	1	4.125
cd	1	6	1	3.833333333
DockingApproach	13	1410	11	1.853754941
DragonAll	13	22	1	19.47619048
DragonAll2	13	27	1	20.77272727
durationThm1	5	7	1	3.333333333
ex3	2	5	1	3.6
ex8	2	5	1	3.4
fast_1	14	19	1	4.166666667
fast_2	14	30	1	4.37037037
FireFly	9	17	1	9.125
Gas	2	8	1	2.444444444
HysteresisAll	2	5	1	5.4
IllinoisAll	10	16	1	11.85714286
Infusion	20	861	5	2.745823389
MesiAll	4	10	1	5.545454545
Metros1	3	16	1	3.533333333
Microwave01	13	126	1	6.417647059
MoesiAll	5	12	1	4.071428571
PetersonAll	12	28	1	14.65517241
ProducerConsumerAll	4	12	1	3.153846154
ProductionCell	3	15	1	3.214285714
Readwrit	9	24	1	12.04
RtpAll	12	24	1	15.96
Speed2	2	5	1	3.6
Stalmark	1	3	1	21
SteamBoilerNoArr1	33	99	1	14.85
SteamBoilerNoArr2	19	3	1	30.66666667
Swimmingpool1	8	21	1	8.1875
Switch	3	2	1	3.333333333
Switch2	3	2	1	3.333333333
SynapseAll	4	10	1	4.555555556
Ticket3iAll	13	20	1	11.45454545
Traffic	1	3	1	5.666666667
Tramway	4	23	1	2.727272727
TwistedCounters	1	4	1	5
Two Counters	1	3	1	2
UMS	5	39	1	2.837837838

systems from the Benchmarks dataset[1]. Several of these models have been used in previous work, including a NASA example, *Docking_Approach*, which describes the behavior of a space shuttle as it docks with the International Space Station [25]. Two other systems, *Infusion_Mgr* and *Alarms*—which represent the prescription man-

[1]Available from `https://github.com/Greg4cr/Reworked-Benchmarks/tree/SingleNode`

agement and alarm-induced behavior of an infusion pump device—were also used in previous work [24, 25, 20].

Information related to each system is provided in Table 4.3, where we again list the number of input variables, number of internal variables, and number of output variables. In this case, we lack the original models, and cannot control the level of inlining. Therefore, we also list the average complexity of expressions to give an idea of how inlined each model is.

4.2 Mutant Generation

Table 4.4: Mutants Information for Rockwell Models (Inlined Versions)

Model	#Total	#Boolean	#Const	#Delay	#Not	#VarReplace	#Arithmetic	#Relational
DWM1	499	4	117	187	74	67	21	29
DWM2	500	51	30	181	154	64	0	20
Latctl_Batch	500	45	44	184	159	49	0	19
Microwave	500	31	20	180	161	106	0	2
Vertmax	499	64	22	181	163	52	0	17

Table 4.5: Mutants Information for Rockwell Models (Non-Inlined Versions)

Model	#Total	#Boolean	#Const	#Delay	#Not	#VarReplace	#Arithmetic	#Relational
DWM1	500	1	90	204	61	103	0	41
DWM2	501	24	29	181	156	103	0	8
Latctl_Batch	500	31	20	178	164	106	0	2
Microwave	500	11	46	193	134	99	7	10
Vertmax	501	51	10	171	164	103	0	2

We have created *mutants* (faulty implementations) for each case example by automatically introducing a single fault into the correct implementation. Each fault was seeded by either inserting a new operator into the system or by replacing an existing operator or variable with a different operator or variable. The following mutation operators were used in this study:

- **Arithmetic:** Changes an arithmetic operator (+, -, /, *, mod, exp).
- **Relational:** Changes a relational operator ($=, \neq, <, >, \leq, \geq$).

Table 4.6: Mutants Information for Benchmark Models

Model	#Total	#Boolean	#Const	#Delay	#Not	#VarReplace	#Arithmetic	#Relational
6counter	118	7	5	44	49	13	0	0
AlarmFunctionalR2012	500	7	92	200	69	89	2	41
CarAll	210	14	18	82	55	30	3	8
cd	159	8	20	59	40	19	2	11
DockingApproach	500	19	53	200	85	101	0	42
DragonAll	502	41	15	212	136	80	11	7
DragonAll2	501	42	15	211	135	80	10	8
durationThm1	155	8	12	66	34	26	4	5
ex3	120	4	13	46	34	18	1	4
ex8	118	4	13	45	33	18	1	4
fast_1	517	41	25	180	201	68	0	2
fast_2	797	66	37	278	307	105	0	4
FireFly	499	36	18	217	121	84	13	10
Gas	173	6	20	73	40	24	6	4
HysteresisAll	172	10	16	65	50	24	1	6
IllinoisAll	500	37	17	216	127	83	13	7
Infusion	501	21	47	204	89	108	0	32
MesiAll	449	17	27	214	79	89	14	9
Metros1	398	14	38	158	109	58	7	14
Microwave01	500	37	35	191	152	69	5	11
MoesiAll	502	14	27	243	80	107	25	6
PetersonAll	500	40	19	213	133	80	8	7
ProducerConsumerAll	318	18	16	142	68	60	7	7
ProductionCell	311	22	7	113	120	49	0	0
Readwrit	500	36	25	217	119	82	10	11
RtpAll	500	45	17	204	144	76	6	8
Speed2	120	4	13	46	34	18	1	4
Stalmark	128	14	3	45	48	18	0	0
SteamBoilerNoArr1	500	34	70	186	97	67	4	42
SteamBoilerNoArr2	420	39	51	147	93	48	0	42
Swimmingpool1	501	29	23	224	113	90	14	8
Switch	81	2	5	28	33	13	0	0
Switch2	81	2	5	28	33	13	0	0
SynapseAll	343	12	19	162	55	71	15	9
Ticket3iAll	500	43	17	207	140	80	3	10
Traffic	100	6	10	40	20	14	2	8
Tramway	424	29	19	149	168	59	0	0
TwistedCounters	165	11	11	66	57	16	1	3
Two Counters	51	1	7	21	15	4	1	2
UMS	813	64	23	285	308	133	0	0

- **Boolean:** Changes a boolean operator ($\vee, \wedge$, XOR).
- **Negation:** Introduces the boolean $\neg$ operator.
- **Delay:** Introduces the delay operator on a variable reference (that is, use the stored value of the variable from the previous computational cycle rather than the newly computed value).
- **Constant:** Changes a constant expression by adding or subtracting 1 from int and real constants, or by negating boolean constants.
- **Variable Replacement:** Substitutes a variable occurring in an equation with another variable of the same type.

The mutation operators used in this study are discussed at length in [51]. This method is designed such that all mutants produced are both syntactically and semantically valid. That is, the mutants will compile, and no mutant will "crash" the system under test.

The type of faults used to create mutants may impact the effectiveness of the selected oracle data when used to test the actual system under test. Note that the type of mutants used in the evaluation in this report are similar to those used by Andrews et al., where the authors found that generated mutants are a reasonable substitute for actual failures in testing experiments [4]. Additionally, recent work from Just et al. suggests a significant correlation between mutant detection and real fault detection [37]. This offers evidence that mutation-based techniques will be useful for supporting the creation of oracles for real-world systems.

In order to control experiment costs, we do not use all possible mutants for each model. Instead, we employ the following rule-of-thumb—if a model has fewer than 500 possible mutations, we use all possible mutations. If over 500 mutations are possible, we choose 500 of them for use in the experiment. In order to select mutants, we first gather a list of all possible mutations. Then, we use the proportions of each mutation type in the full set to select the number of mutants for the reduced set of 500, or a little bit greater than 500 due to some calculating error. The only violation is ums.lus, for which we generated the whole pool of 813 mutations as it's stuck when trying to generate a reduced set. Mutants of each type are then chosen randomly until the determined number are chosen for that type. This process prevents biasing towards particular types of mutations. Instead, the proportion of each fault type is maintained, despite not using the full set of mutations. In Tables 4.5, 4.4, and 4.6, we list the number of mutants used for each model from the Rockwell Collins (non-inlined and inlined variants) and Benchmarks datasets.

4.3 Test Data Generation

In this research, we explore four structural coverage criteria: Condition Coverage, Decision Coverage, Branch Coverage, and Modified Condition/Decision Coverage (MC/DC) [35, 8]. These criteria are defined in Section 2.2.

For each criterion, we generate tests for both the traditional criterion as well as a version requiring observability. We refer to the observable versions of each criterion as Observable Condition Coverage (**OCondition**), Observable Decision Coverage (**ODecision**), Observable Branch Coverage (**OBranch**), and Observable MC/DC (**OMC/DC**).

For our directed test generation approach, we used counterexample-based test generation to generate tests satisfying the four coverage criteria and their observable variants [19, 52]. In this approach, each coverage obligation is encoded as a temporal logic formula in the model, and a model checker is used to produce a counterexample illustrating how the coverage obligation can be covered. This counterexample offers test input—a series of values for each input variable for one or more test steps. By repeating this process for each coverage obligation for the system, we can use the model checker to derive test sequences intended to achieve the maximum possible coverage of the model. We have used the JKind model checker [27, 18] in our experiments because we have found that it is efficient and produces tests that are easy to understand [31].

4.4 Test Suite Reduction

Counterexample-based test generation results in a separate test for each coverage obligation. This leads to a large amount of redundancy in the tests generated, as each test likely covers several obligations. Consequently, the test suite generated for each coverage criterion is generally much larger than is required to provide coverage.

Given the correlation between test suite size and fault finding effectiveness [47], this has the potential to yield misleading results—an unnecessarily large test suite may lead us to conclude that a coverage criterion has led us to select effective tests, when in reality it is the size of the test suite that is responsible for its effectiveness. To avoid this, we reduce each naïvely generated test suite while maintaining the coverage achieved. To prevent us from selecting a test suite that happens to be exceptionally good or exceptionally poor relative to the possible reduced test suites, we produce 50 different reduced test suites for each case example using the process described below.

Reduction is performed using a simple greedy algorithm. We determine the coverage obligations satisfied by each test generated, and initialize an empty test set *reduced*. We then randomly select a test from the full set of tests; if it satisfies obligations not satisfied by any test input in *reduced*, we add it to *reduced*. We continue until all tests have been examined in the full set of tests.

When generating tests suites to satisfy a structural coverage criterion, the suite size can vary from the minimum required to satisfy the coverage criterion (generally unknown) to infinity. Previous work has demonstrated that test suite reduction can have a negative impact on test suite effectiveness [30]. Despite this, we believe the test suite size most likely to be used in practice is one designed to be small—reduced with respect to coverage—rather than large (every test generated in the case of counterexample-based generation or, even more arbitrarily, 1,000 random tests). Note that one could build a counterexample-based test suite generation tool that, upon generating a test, removes from consideration *all* newly covered obligations, and randomly selects a new uncovered obligation to try to satisfy, repeating until finished. Such a tool would produce test suites equivalent to our reduced test suites, and thus require no reduction; alternatively, we could view such test suites as pre-reduced.

4.5 Computing Effectiveness

In order to compute effectiveness of the generated test suites, we produce *traces* of execution by executing each test case against the original program and each mutant—recording the value of all variables at each step.

In our study, we use what are known as *expected value oracles* as our test oracles [24]. Consider the following testing process for a software system: (1) the tester selects inputs using some criterion—structural coverage, random testing, or engineering judgment; (2) the tester then defines concrete, anticipated values for these inputs for one or more variables (internal variables or output variables) in the program. Past experience with industrial practitioners indicates that such oracles are commonly used in testing critical systems, such as those in the avionics or medical device fields.

We explore the use of two types of expected value oracles: an *output-only oracle* that defines expected values for all outputs, and a *maximum oracle* that defines expected values for all outputs and all internal state variables. The output-only oracle represents the oracle most likely to be used in practice. Both oracles have been used in previous work, and thus we use both to allow for comparison [23, 24].

To produce and oracle, we use the values of the monitored variables from the traces gathered by executing test cases on the original program, and we compare those values to those recorded for each mutant. The fault finding effectiveness of the test suite and oracle pair is computed as the number of mutants detected (or "killed"). For all studied systems, we assess the fault-finding effectiveness of each test suite and oracle combination by calculating the ratio of mutants killed to total number of mutants.

Test generation and trace generation are performed using in-house automation. This framework is open-source and freely available from the project repository: `https://github.com/MENG2010/lustre`.

Chapter 5

Results and Discussion

In this section, we will address our research questions and discuss the implications of the results. As a reminder, we are interested in the following:

1. Which criterion has the highest average likelihood of fault detection? (Section 5.1)

2. Are test suites generated to satisfy observable variants of coverage criteria more effective than the test suites generated to satisfy the original criterion? (Section 5.2)

3. What impact does observability have on the average size of the generated test suites and the average percentage of satisfied obligations for each criterion? (Section 5.3)

4. Across the studied criteria, does observability have a consistent effect on efficacy in terms of factors such as likelihood of fault detection, oracle and structure sensitivity, and satisfiability of obligations? (Section 5.4)

5.1 Overall Efficacy

Table 5.5 lists the average percentage of faults detected by test suites generated for each of the eight coverage criteria, separated by oracle type, for the Rockwell and Benchmarks datasets. Tables 5.1-5.4 list the average results for each individual model from each dataset. These results are also plotted in Figure 5.1. From these results,

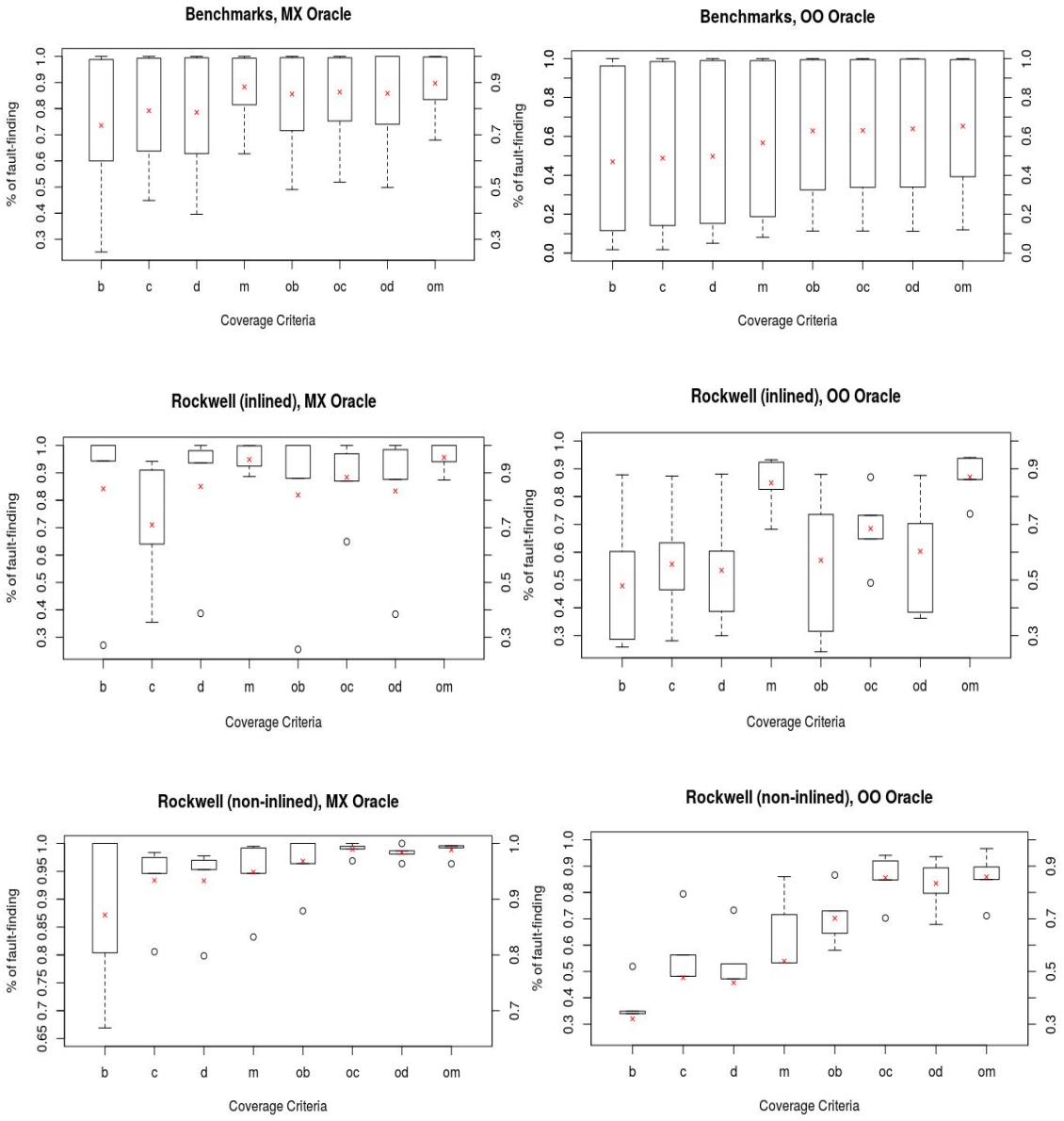

Figure 5.1: Percent of mutants killed for each combination of test suite and oracle. b = Branch Coverage, c = Condition, d = Decision, m = MC/DC, o = Observable.

Table 5.1: Percentage of mutants killed for each combination of Rockwell (inlined) model and oracle.

	Model	Branch	Obranch	Condition	OCondition	Decision	ODecision	MC/DC	OMC/DC
MX Oracle	DWM1	99.60%	88.00%	94.23%	87.00%	93.62%	87.60%	88.66%	87.40%
	DWM2	27.09%	25.60%	64.02%	64.90%	38.71%	38.43%	93.28%	94.06%
	Latctl_Batch	100.00%	100.00%	70.30%	93.04%	98.09%	92.42%	100.00%	100.00%
	Microwave	94.31%	95.98%	90.99%	96.96%	94.74%	98.46%	92.45%	96.57%
	Vertmax	100.00%	100.00%	35.47%	100.00%	100.00%	100.00%	99.87%	100.00%
OO Oracle	DWM1	87.84%	88.00%	87.40%	87.00%	88.11%	87.60%	88.20%	87.40%
	DWM2	25.89%	24.20%	63.38%	64.79%	38.71%	38.43%	93.28%	94.05%
	Latctl_Batch	60.27%	73.56%	46.50%	68.48%	60.38%	69.06%	82.61%	86.15%
	Microwave	36.86%	68.19%	53.14%	73.25%	50.21%	70.30%	68.24%	73.78%
	Vertmax	28.69%	31.55%	28.14%	48.97%	29.98%	36.26%	92.37%	93.77%

Table 5.2: Average percentage of mutants killed for each combination of Rockwell (non-inlined) model and oracle.

	Model	Branch	Obranch	Condition	OCondition	Decision	ODecision	MC/DC	OMC/DC
MX Oracle	DWM1	80.39%	96.39%	80.58%	96.89%	79.84%	96.38%	83.23%	96.36%
	DWM2	66.91%	87.90%	98.38%	99.03%	97.78%	98.69%	99.49%	99.60%
	Latctl_Batch	100.00%	100.00%	97.48%	99.49%	96.65%	98.66%	99.20%	99.24%
	Microwave	88.57%	99.89%	94.63%	99.35%	95.32%	98.10%	94.64%	99.47%
	Vertmax	100.00%	100.00%	95.82%	100.00%	96.99%	100.00%	97.78%	99.60%
OO Oracle	DWM1	4.94%	86.60%	4.77%	86.91%	4.66%	86.70%	5.04%	86.93%
	DWM2	34.91%	68.84%	79.43%	94.10%	73.26%	93.61%	86.01%	96.69%
	Latctl_Batch	51.90%	73.01%	56.26%	84.77%	52.90%	79.68%	71.58%	84.92%
	Microwave	34.50%	64.48%	49.57%	70.28%	50.63%	67.85%	53.26%	71.16%
	Vertmax	33.99%	58.05%	48.17%	91.97%	47.21%	89.36%	53.54%	89.70%

we can see that—on average—test suites generate to satisfy OMC/DC tend to kill a larger percent of mutants than test suites satisfying all other coverage criteria. For both variants of the Rockwell systems—with any oracle—test suites generated to satisfy OMC/DC kill the most mutants. The sole exception is for the non-inlined variant—with the maximum oracle—where OCondition suites outperform OMC/DC by 0.1%. For the Benchmark models—with any oracle—OMC/DC-satisfying suites have the highest overall average possibility of revealing faults.

Test suites satisfying Observable MC/DC are generally the most effective, killing 95.61% of mutants on average (MX oracle) and 87.03% (OO oracle) for the inlined Rockwell models, 98.85% (MX)/85.88% (OO) for the non-inlined Rockwell models, and 89.62% (MX)/65.14% (OO) for the Benchmarks models.

We can examine this question further through statistical analysis. To address

Table 5.3: Average percentage of mutants killed for each Benchmark model with MX oracle. NA means there were no obligations for that criterion for that particular model.

Model	Branch	Obranch	Condition	OCondition	Decision	ODecision	MC/DC	OMC/DC
6counter	NA	NA	99.16%	99.16%	99.16%	99.16%	99.16%	99.16%
AlarmFunctionalR2012	72.51%	74.62%	72.18%	73.73%	72.75%	73.46%	72.87%	74.46%
CarAll	100.00%	100.00%	91.10%	100.00%	89.30%	93.20%	90.43%	85.15%
cd	53.58%	83.02%	84.81%	89.31%	72.38%	72.38%	88.59%	90.57%
DockingApproach	72.37%	89.70%	75.85%	51.87%	75.91%	100.00%	77.63%	74.23%
DragonAll	33.16%	64.67%	44.88%	73.92%	48.43%	69.17%	88.18%	93.72%
DragonAll2	38.25%	62.48%	50.05%	77.65%	39.60%	69.48%	87.06%	91.64%
durationThm1	25.16%	83.23%	71.81%	84.45%	56.13%	83.87%	83.23%	87.74%
ex3	65.83%	69.17%	63.10%	69.13%	61.67%	68.07%	68.33%	68.33%
ex8	72.03%	75.42%	69.47%	74.54%	66.37%	74.58%	72.88%	75.22%
fast_1	78.60%	96.15%	95.64%	100.00%	79.23%	100.00%	100.00%	100.00%
fast_2	80.41%	100.00%	92.08%	96.10%	85.41%	100.00%	96.54%	96.74%
FireFly	63.76%	71.40%	59.38%	76.07%	60.98%	72.76%	81.02%	89.23%
Gas	52.02%	77.62%	73.41%	84.62%	81.92%	79.38%	89.02%	85.48%
HysteresisAll	99.42%	99.42%	99.42%	99.42%	98.81%	99.42%	99.42%	99.42%
IllinoisAll	42.48%	71.60%	64.05%	79.47%	46.83%	74.84%	87.18%	89.85%
Infusion	61.84%	82.10%	70.08%	84.61%	69.65%	82.12%	73.31%	84.04%
MesiAll	59.99%	99.33%	66.45%	77.67%	60.20%	87.81%	62.74%	76.17%
Metros1	96.65%	99.50%	99.50%	99.50%	99.50%	99.50%	99.50%	99.50%
Microwave01	48.96%	95.80%	76.22%	87.92%	78.80%	91.73%	98.80%	96.08%
MoesiAll	69.73%	69.35%	60.25%	62.32%	63.97%	57.99%	62.87%	67.98%
PetersonAll	53.48%	87.67%	60.81%	73.41%	65.83%	82.72%	86.10%	76.02%
ProducerConsumerAll	100.00%	100.00%	92.09%	96.35%	100.00%	100.00%	97.18%	100.00%
ProductionCell	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Readwrit	72.42%	93.21%	83.74%	88.08%	81.86%	94.88%	98.00%	95.28%
RtpAll	98.80%	95.20%	79.90%	97.66%	87.38%	95.05%	98.80%	96.96%
Speed2	65.48%	69.17%	63.05%	69.02%	61.67%	68.17%	67.50%	68.33%
Stalmark	NA	NA	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
SteamBoilerNoArr1	83.24%	99.60%	78.60%	97.08%	99.40%	99.40%	95.96%	99.40%
SteamBoilerNoArr2	44.29%	49.05%	60.08%	58.28%	43.95%	49.85%	84.76%	85.53%
Swimmingpool1	75.97%	74.31%	71.76%	79.00%	71.43%	75.54%	84.91%	86.24%
Switch	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Switch2	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
SynapseAll	62.10%	62.10%	60.92%	63.06%	58.68%	58.60%	64.02%	64.14%
Ticket3iAll	98.21%	100.00%	61.97%	100.00%	100.00%	100.00%	94.54%	100.00%
Traffic	68.00%	68.00%	83.46%	87.14%	75.76%	76.00%	86.00%	90.00%
Tramway	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
TwistedCounters	99.39%	99.39%	99.39%	98.23%	99.39%	99.39%	98.08%	98.08%
Two Counters	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
UMS	NA	NA	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%

this, we first formulate our hypothesis as follow:

H_1: For each system in our study—with any oracle—the OMC/DC criterion produces test suites with the highest likelihood of fault detection.

The paired null hypothesis is,

H_θ: For each system in our study—with any oracle—the OMC/DC criterion produces test suites with a likelihood of fault detection drawn from the same distribution as another criterion's suites.

We have performed a one-sided (strictly greater) Mann-Whiteney-Wilconxon rank-

Table 5.4: Average percentage of mutants killed for each Benchmark model with OO oracle. NA means there were no obligations for that criterion for that particular model.

Model	Branch	Obranch	Condition	OCondition	Decision	ODecision	MC/DC	OMC/DC
6counter	NA	NA	99.16%	99.16%	99.16%	99.16%	99.16%	99.16%
AlarmFunctionalR2012	29.59%	44.75%	29.30%	44.22%	29.21%	45.01%	28.02%	45.48%
CarAll	98.28%	100.00%	54.88%	100.00%	57.07%	73.00%	50.20%	26.40%
cd	28.10%	48.43%	51.14%	56.60%	40.57%	41.56%	53.37%	57.23%
DockingApproach	33.07%	70.01%	33.42%	25.82%	35.35%	100.00%	37.49%	60.02%
DragonAll	1.80%	22.71%	1.80%	26.54%	5.08%	23.35%	9.91%	35.36%
DragonAll2	6.00%	28.54%	12.80%	36.09%	6.00%	25.87%	16.89%	40.53%
durationThm1	11.61%	37.47%	16.80%	44.17%	14.19%	39.14%	20.65%	38.06%
ex3	30.83%	32.50%	28.38%	32.47%	25.00%	31.53%	31.67%	31.67%
ex8	65.25%	69.49%	62.88%	68.61%	36.44%	66.10%	66.10%	69.29%
fast_1	47.21%	89.04%	86.68%	100.00%	40.74%	100.00%	100.00%	100.00%
fast_2	61.30%	100.00%	80.79%	89.14%	64.11%	100.00%	86.44%	88.46%
FireFly	11.42%	30.00%	9.08%	34.40%	10.82%	35.44%	10.10%	45.80%
Gas	12.14%	50.87%	15.61%	53.99%	51.45%	51.61%	69.36%	71.68%
HysteresisAll	99.42%	99.42%	99.42%	99.42%	98.73%	99.42%	99.42%	99.42%
IllinoisAll	5.00%	31.60%	6.28%	35.06%	7.60%	31.80%	11.85%	42.13%
Infusion	11.59%	27.23%	8.54%	30.28%	8.98%	27.22%	10.23%	29.81%
MesiAll	16.48%	99.33%	26.79%	57.85%	16.37%	77.33%	14.56%	58.70%
Metros1	87.36%	99.50%	99.50%	99.50%	99.50%	99.50%	99.50%	99.50%
Microwave01	8.60%	14.00%	8.70%	14.00%	9.00%	13.40%	9.80%	14.28%
MoesiAll	58.96%	58.69%	32.38%	46.98%	43.31%	45.24%	34.61%	55.03%
PetersonAll	8.54%	63.69%	8.68%	19.17%	20.62%	50.76%	8.71%	19.44%
ProducerConsumerAll	9.06%	12.58%	6.24%	13.52%	6.71%	13.52%	14.47%	14.47%
ProductionCell	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Readwrit	34.22%	81.92%	52.74%	56.55%	58.57%	85.88%	98.00%	87.96%
RtpAll	98.80%	90.73%	61.24%	95.55%	79.44%	89.10%	98.80%	93.99%
Speed2	31.67%	33.33%	29.70%	33.23%	25.00%	32.42%	32.50%	32.50%
Stalmark	NA	NA	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
SteamBoilerNoArr1	7.55%	11.36%	7.32%	11.38%	6.60%	11.25%	8.15%	11.98%
SteamBoilerNoArr2	44.29%	49.05%	59.84%	58.28%	43.71%	49.85%	84.76%	85.53%
Swimmingpool1	4.57%	62.20%	6.30%	62.91%	53.99%	59.35%	54.01%	64.07%
Switch	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Switch2	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
SynapseAll	43.15%	43.15%	28.88%	42.86%	9.61%	40.82%	33.07%	44.90%
Ticket3iAll	96.17%	100.00%	16.30%	100.00%	100.00%	100.00%	60.57%	100.00%
Traffic	19.00%	19.00%	31.46%	32.86%	19.00%	30.00%	37.00%	45.00%
Tramway	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
TwistedCounters	99.39%	99.39%	99.39%	96.65%	99.39%	99.39%	97.04%	97.95%
Two Counters	100.00%	100.00%	97.84%	100.00%	98.82%	100.00%	98.63%	100.00%
UMS	NA	NA	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%

sum test [66], a non-parametric hypothesis test used to determining whether two independent samples were selected from populations having the same distribution, to verify our hypothesis. Since we cannot generalize across non-randomly selected case examples, we apply the statistical test over various pairs of coverage criteria (i.e., any of the coverage criteria versus the rest of the coverage criteria respectively, therefore, we have 56 pairs of metrics in total), for each pairing of model and oracle type, with $\alpha = 0.05$.

The statistical results are presented in Table 5.6. In this table, we list the percent-

Table 5.5: Average percentage of mutants killed for each pairing of criterion and oracle over all models in each dataset.

	Rockwell (Inlined)		Rockwell (Non-inlined)		Benchmark	
	MX	OO	MX	OO	MX	OO
OMC/DC	95.61%	87.03%	98.85%	85.88%	89.62%	65.14%
OCondition	88.38%	68.50%	98.95%	85.61%	86.22%	62.93%
ODecision	83.38%	60.33%	98.37%	83.44%	86.21%	64.70%
OBranch	81.92%	57.10%	96.84%	70.19%	85.47%	62.70%
MC/DC	94.85%	84.94%	94.87%	53.89%	88.36%	57.13%
Condition	71.00%	55.71%	93.38%	47.64%	79.37%	50.50%
Decision	85.03%	53.48%	93.32%	45.73%	78.81%	49.26%
Branch	84.20%	47.91%	87.17%	32.05%	73.19%	46.50%

age of cases for each dataset where we can reject H_θ—that is, where we can confirm that OMC/DC outperforms the compared criterion. We also list the percentage of cases where the reverse is true—where we can state that the other criterion outperforms OMC/DC with significance. For example, for the Rockwell (Non-inlined) models, with an output-only oracle, OMC/DC outperforms all criteria except OCondition in 100% of cases, with statistical significance.

Table 5.6: Percent of cases where OMC/DC suites outperform suites satisfying other criteria with significance, and percent of cases where suites satisfying the other criteria outperform OMC/DC suites.

		MX Oracle		OO Oracle	
		more effective	less effective	more effective	less effective
Benchmark	ODecision	45.00%	10.00%	45.00%	15.00%
	OCondition	52.50%	15.00%	47.50%	15.00%
	OBranch	37.84%	28.95%	45.95%	21.05%
	MC/DC	45.00%	15.00%	55.00%	5.00%
	Decision	57.50%	2.50%	67.50%	2.50%
	Condition	65.00%	5.00%	72.50%	2.50%
	Branch	65.79%	7.69%	71.05%	7.69%
Rockwell (Inlined)	ODecision	40.00%	40.00%	80.00%	20.00%
	OCondition	60.00%	0.00%	100.00%	0.00%
	OBranch	20.00%	20.00%	80.00%	20.00%
	MC/DC	40.00%	20.00%	80.00%	20.00%
	Decision	40.00%	20.00%	80.00%	20.00%
	Condition	80.00%	20.00%	80.00%	0.00%
	Branch	20.00%	20.00%	80.00%	20.00%
Rockwell (Non-inlined)	ODecision	80.00%	20.00%	100.00%	0.00%
	OCondition	20.00%	20.00%	40.00%	20.00%
	OBranch	40.00%	60.00%	100.00%	0.00%
	MC/DC	80.00%	0.00%	100.00%	0.00%
	Decision	100.00%	0.00%	100.00%	0.00%
	Condition	100.00%	0.00%	100.00%	0.00%
	Branch	60.00%	40.00%	100.00%	0.00%

For Benchmarks, with any oracle, the percentage of cases where OMC/DC suites outperform suites satisfying other coverage criteria is always higher than the percentage of suites satisfying other criteria outperforming OMC/DC suites. That is, OMC/DC always has a highest average likelihood of fault detection. This is also true in all situations for both variants of the Rockwell models with an output-only oracle. Results are a little less clear-cut for the Rockwell models when paired with a maximum oracle, where other criteria occasionally tie or outperform OMC/DC. For instance, for the inlined variants, ODecision, OBranch, and Branch suites outperform OMC/DC suites as often as OMC/DC suites outperform their counterparts.

Intuitively, these results makes sense. There is a clear boost in performance from the addition of observability. As Table 5.5 shows, the observable versions of criteria almost always outperform both their non-observable counterpart and *all other non-observable criteria*, except the original MC/DC. MC/DC suites outperform all of the other non-observable versions of the studied criteria, and is the only non-observable criterion to produce suites that occasionally outperform the observable counterparts. The addition of observability boosts the efficacy of the generated test suites, generally with the end result that Observable MC/DC produces the most effective test suites. OMC/DC does not always produce the best suites, but it is the safest choice of any of the studied criteria.

Across the board, efficacy tends to be higher for the maximum oracle, and the gap between observable and non-observable criteria tends to be less. This can be explained by examining the concept of masking. With an output-only oracle, input must trigger a fault, and the effect of a fault must not be masked by expressions on the path to the output. Observability is intended to overcome masking, and clearly does assist—given the results for output-only oracles. However, with a maximum oracle, we already have expression-level observability. Masking along the path to the output does not need to be overcome. The observable criteria generally produce more

effective suites even in these cases, but the possibility for improvement is smaller.

In general, however, maximum oracles are prohibitively expensive to employ [24]. A tester would need to specify expected values for all variables, for each test step. This is not usually a realistic goal. Output-only oracles are the most common, and OMC/DC appears to be the most effective criterion when paired with this common oracle.

5.2 Efficacy Impact of the Addition of Observability

Table 5.7: Average improvement in mutant detection for each combination of Rockwell (inlined) model and oracle

	Model	Branch	Condition	Decision	MC/DC
MX Oracle	DWM1	-11.65%	-7.67%	-6.43%	-1.42%
	DWM2	-5.51%	1.37%	-0.71%	0.84%
	Latctl_Batch	0.00%	32.36%	-5.78%	0.00%
	Microwave	1.77%	6.56%	3.93%	4.45%
	Vertmax	0.00%	181.95%	0.00%	0.13%
OO Oracle	DWM1	0.18%	-0.46%	-0.58%	-0.91%
	DWM2	-6.53%	2.23%	-0.71%	0.83%
	Latctl_Batch	22.05%	47.26%	14.38%	4.29%
	Microwave	84.98%	37.85%	40.01%	8.11%
	Vertmax	9.97%	74.03%	20.95%	1.52%

Table 5.8: Average improvement in mutant detection for each combination of Rockwell (non-inlined) model and oracle

	Model	Branch	Condition	Decision	MC/DC
MX Oracle	DWM1	19.91%	20.23%	20.72%	15.77%
	DWM2	31.38%	0.67%	0.92%	0.12%
	Latctl_Batch	0.00%	2.06%	2.09%	0.04%
	MicrowaveI	12.78%	4.99%	2.92%	5.10%
	Vertmax	0.00%	4.36%	3.10%	1.86%
OO Oracle	DWM1	1654.38%	1722.82%	1760.52%	1623.39%
	DWM2	97.18%	18.47%	27.77%	12.42%
	Latctl_Batch	40.69%	50.67%	50.62%	18.62%
	Microwave	86.91%	41.78%	34.02%	33.60%
	Vertmax	70.79%	90.93%	89.28%	67.54%

In Table 5.10, we present the average improvement in efficacy when moving from a traditional criterion—such as MC/DC—to its observable counterpart over all models for each dataset. Tables 5.7-5.9 list the average improvement for each individual

Table 5.9: Average improvement in mutant detection for each combination of Benchmark model and oracle

Model	MX Oracle				OO Oracle			
	Branch	Condition	Decision	MC/DC	Branch	Condition	Decision	MC/DC
6counter	N/A	0.00%	0.00%	0.00%	N/A	0.00%	0.00%	0.00%
AlarmFunctionalR2012	2.91%	2.14%	0.98%	2.18%	51.24%	50.91%	54.07%	62.35%
CarAll	0.00%	9.78%	4.37%	-5.83%	1.75%	82.23%	27.92%	-47.41%
cd	54.93%	5.31%	0.00%	2.23%	72.34%	10.67%	2.45%	7.24%
DockingApproach	23.96%	-31.62%	31.73%	-4.38%	111.73%	-22.75%	182.87%	60.10%
DragonAll	95.05%	64.72%	42.83%	6.27%	1161.78%	1374.67%	359.97%	256.78%
DragonAll2	63.35%	55.16%	75.43%	5.26%	375.73%	182.03%	331.20%	140.00%
durationThm1	230.77%	17.61%	49.43%	5.43%	222.67%	162.90%	175.73%	84.37%
ex3	5.06%	9.56%	10.38%	0.00%	5.41%	14.39%	26.13%	0.00%
ex8	4.71%	7.29%	12.36%	3.21%	6.49%	9.11%	81.40%	4.82%
fast_1	22.33%	4.56%	26.21%	0.00%	88.58%	15.36%	145.49%	0.00%
fast_2	24.36%	4.37%	17.09%	0.21%	63.12%	10.34%	55.97%	2.34%
FireFly	11.98%	28.11%	19.33%	10.13%	162.79%	279.02%	227.42%	353.47%
Gas	49.20%	15.28%	-3.10%	-3.97%	319.05%	245.93%	0.31%	3.33%
HysteresisAll	0.00%	0.00%	0.61%	0.00%	0.00%	0.00%	0.69%	0.00%
IllinoisAll	68.53%	24.07%	59.81%	3.06%	532.00%	457.86%	318.64%	255.57%
Infusion	32.77%	20.74%	17.90%	14.63%	135.00%	254.61%	202.98%	191.32%
MesiAll	65.58%	16.88%	45.87%	21.41%	502.87%	115.96%	372.41%	303.27%
Metros1	2.95%	0.00%	0.00%	0.00%	13.90%	0.00%	0.00%	0.00%
Microwave01	95.65%	15.36%	16.41%	-2.76%	62.79%	60.85%	48.89%	45.67%
MoesiAll	-0.54%	3.44%	-9.35%	8.13%	-0.45%	45.09%	4.44%	58.97%
PetersonAll	63.91%	20.72%	25.65%	-11.70%	645.76%	120.98%	146.10%	123.24%
ProducerConsumerAll	0.00%	4.62%	0.00%	2.90%	38.79%	116.73%	101.50%	0.00%
ProductionCell	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Readwrit	28.71%	5.18%	15.91%	-2.78%	139.41%	7.22%	46.64%	-10.24%
RtpAll	-3.64%	22.23%	8.78%	-1.86%	-8.17%	56.02%	12.16%	-4.87%
Speed2	5.62%	9.46%	10.54%	1.23%	5.26%	11.90%	29.67%	0.00%
Stalmark	N/A	0.00%	0.00%	0.00%	N/A	0.00%	0.00%	0.00%
SteamBoilerNoArr1	19.65%	23.52%	0.00%	3.59%	50.45%	55.49%	70.42%	46.98%
SteamBoilerNoArr2	10.75%	-3.00%	13.41%	0.90%	10.75%	-2.61%	14.03%	0.90%
Swimmingpool1	-2.19%	10.09%	5.76%	1.57%	1259.67%	898.10%	9.92%	18.63%
Switch	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Switch2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
SynapseAll	0.00%	3.50%	-0.13%	0.19%	0.00%	48.39%	324.76%	35.78%
Ticket3iAll	1.82%	61.37%	0.00%	5.78%	3.98%	513.35%	0.00%	65.10%
Traffic	0.00%	4.41%	0.32%	4.65%	0.00%	4.45%	57.89%	21.62%
Tramway	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
TwistedCounters	0.00%	-1.17%	0.00%	0.00%	0.00%	-2.76%	0.00%	0.94%
Two Counters	0.00%	0.00%	0.00%	0.00%	0.00%	2.20%	1.19%	1.39%
UMS	N/A	0.00%	0.00%	0.00%	N/A	0.00%	0.00%	0.00%

model. In general, we see an increase in efficacy. That is, regardless of the underlying coverage criterion, observability seems to have a positive impact on the likelihood of detecting faults.

This is especially true when an output-only oracle—the most common oracle [24]—is used. When using an output-only oracle, masking is an especially prevalent problem. Observability is intended to overcome masking, and our results show that this is the case. This can be clearly seen in the Rockwell (non-inlined) models, where the addition of observability improves efficacy up to 392.44%. Results are more subdued

Table 5.10: Average improvement in the likelihood of fault detection, after adding observability constraints

		MX Oracle	**OO Oracle**
Benchmark	MC/DC	1.79%	53.38%
	Decision	12.78%	88.03%
	Condition	11.12%	132.79%
	Branch	26.44%	163.10%
Rockwell (Inlined)	MC/DC	0.80%	2.77%
	Decision	-1.80%	14.81%
	Condition	42.92%	32.18%
	Branch	-3.08%	22.13%
Rockwell (Non-inlined)	MC/DC	4.58%	351.12%
	Decision	5.95%	392.44%
	Condition	6.46%	384.93%
	Branch	12.81%	389.99%

for the inlined variants—up to a 32.18% improvement.

We can see from these results that the structure of the system—how code is written—has some impact on the impact of adding observability. The Rockwell examples offer two extremes—either entirely inlined or with the simplest possible expressions. At the later end, there is tremendous improvement from adding observability. If there are a large number of simple expressions, then masking along the path to the output is far more likely than if there are a smaller number of expressions. As a result, observability has a major impact, propagating the effect of a fault to the output variables. On the other hand, if there are a small number of expressions, then the path to output will be shorter. Therefore, observability will have a smaller impact.

In addition, past work has shown that the test cases generated for heavily inlined systems may be more effective from the start [23, 50]. Criteria such as MC/DC require that an independent impact be shown for each condition within a decision. That is, if MC/DC is fulfilled, then a condition will not be masked within the expression that it appears in. Its impact can be masked on the path to output, but will effect the outcome of the decision that it falls within. If a model is more heavily inlined, then the requirements of standard MC/DC are more strenuous—independent impact must be shown for more complex expressions. At the same time, the path to output

is shorter, limiting further opportunities for masking. Therefore, the test cases may be more effective from the start, and the further impact of observability may be more limited.

We can see some evidence from this that observability helps bridge the gap from output-only oracle to maximum oracle—without adding additional human oracle cost [45], and the gap from non-inlined to inlined program structure. The Benchmarks examples are varied in terms of structure. As a result, the impact of adding observability falls between the two extremes of the Rockwell models—with improvements of up to 163.10% for the output-only oracle.

As noted earlier, improvements tend to be smaller when employing a maximum oracle. For non-inlined implementation of Rockwell models, we see average improvements of up to 12.81%. For the inlined variants, we see up to a 42.92% average improvement, and even see small performance downgrades of up to 3.08%. For the Benchmarks dataset, we see average improvements of up to 26.44%.

Adding observability tends to improve efficacy over satisfaction of the traditional criteria, with average improvements of up to 392.44% in mutation detection and per-model improvements of up to 1654.38%.

We can establish evidence by performing statistical analysis, employing the same test used previously. We formulate our hypotheses as follow:

H_2: For each system in our study—with any oracle—the observable version of a criterion produces test suites with a higher likelihood of fault detection than the traditional variant.

The paired null hypothesis is:

$H_{\theta 2}$: For each system in our study—with any oracle—the observable version of a criterion produces test suites with a likelihood of fault detection drawn from the

same distribution as the traditional variant.

Table 5.11: Cases where the observable criterion produces suites outperforming the non-observable variant with significance, and when the non-observable variant produces more effective suites.

		MX Oracle		OO Oracle	
		Observable Metric	Traditional Metric	Observable Metric	Traditional Metric
Benchmarks	MC/DC	45.00%	15.00%	55.00%	5.00%
	Decision	57.50%	5.00%	67.50%	2.50%
	Condition	65.00%	7.50%	67.50%	7.50%
	Branch	60.53%	5.26%	71.05%	2.63%
Rockwell (Inlined)	MC/DC	40.00%	20.00%	80.00%	20.00%
	Decision	20.00%	40.00%	60.00%	20.00%
	Condition	60.00%	20.00%	80.00%	20.00%
	Branch	0.00%	40.00%	80.00%	20.00%
Rockwell (Non-inlined)	MC/DC	80.00%	0.00%	100.00%	0.00%
	Decision	100.00%	0.00%	100.00%	0.00%
	Condition	100.00%	0.00%	100.00%	0.00%
	Branch	60.00%	0.00%	100.00%	0.00%

The statistical results are presented in Table 5.11, where we list the percent of cases where we can reject $H_{\theta 2}$—we can provide evidence that the observable criterion produces more effective test suites—along with the percentage of cases where we can state with significance that the reverse is true—that the traditional criterion is more effective. For example, for the Benchmark models—with a maximum oracle—suites satisfying the OMC/DC criterion outperform MC/DC-satisfying suites with significance in 45% of cases, while the reverse is true for only 15% of cases. For the remaining 40% of the models, neither outperforms the other with significance.

Almost universally, the observable variant outperforms the traditional variant—with significance—in more cases. The only two situations where this is reversed are for Decision Coverage and Branch Coverage on the inlined Rockwell models, paired with a maximum oracle. As highlighted above, this is the exact situation where we would expect the least benefit from the addition of observability. However, with the more realistic output-only oracle, the observable variant of the criterion produces more effective suites in the vast majority of cases.

We can further examine the impact of observability by looking at two situations—when observability had the *most* impact on efficacy, and when it had the *least*—or

Table 5.12: Lowest improvement (or worst downgrade) in fault detection when transitioning from traditional to observable criteria.

		MX Oracle	OO Oracle
Benchmark	MC/DC	-11.7%, PetersonAll	-47.41%, Car_All
	Decision	-9.35%, MoesiAll	0.00%, 6counter/Metros1/ProductionCell/ Stalmark/Switch/Switch2/Ticket3iAll/ Tramway/TwistedCounters
	Condition	-31.62%, DockingApproach	-22.75%, DockingApproach
	Branch	-3.64%, Rtp_All	-8.17%, Rtp_All
Rockwell (Inlined)	MC/DC	-1.42%, DWM1	-0.91%, DWM1
	Decision	-6.43%, DWM1	-0.71%, DWM2
	Condition	-7.67%, DWM1	-0.46%, DWM1
	Branch	-11.65%, DWM1	-6.53%, DWM2
Rockwell (Non-inlined)	MC/DC	0.04%, Latctl_Batch	12.42%, DWM2
	Decision	0.92%, DWM2	27.77%, DWM2
	Condition	0.67%, DWM2	18.47%, DWM2
	Branch	0.00%, Latctl_Batch/Vertmax	40.69%, Latctl_Batch

Table 5.13: Max improvement in fault detection when transitioning from traditional to observable criteria.

		MX Oracle	OO Oracle
Benchmark	MC/DC	21.41%, MesiAll	353.47%, FireFly
	Decision	75.43%, DragonAll2	372.41%, MesiAll
	Condition	64.72%, DragonAll	1374.67%, DragonAll
	Branch	230.77%, durationThm1	1259.67%, Swimmingpool1
Rockwell (Inlined)	MC/DC	4.45%, Microwave	8.11%, Microwave
	Decision	3.93%, Microwave	40.01%, Microwave
	Condition	181.95%, Vertmax	74.03%, Vertmax
	Branch	1.77%, Microwave	84.98%, Microwave
Rockwell (Non-inlined)	MC/DC	15.77%, DWM1	1623.39%, DWM1
	Decision	20.72%, DWM1	1760.52%, DWM1
	Condition	20.23%, DWM1	1722.82%, DWM1
	Branch	31.38%, DWM2	1654.38%, DWM1

even *negative*—impact on results.

First, we can examine the "minimal improvements" to examine situations where suites satisfying the observable criteria are worse than their non-observable counterparts. In Table 5.12, we list the smallest improvement seen—or largest loss in efficacy, if there was one—and the affected model, for each criterion and oracle. With an output-only oracle, we see a small loss in performance in the worst cases for the inlined Rockwell models, with drops of 0.56-6.53% in efficacy. For the Benchmarks models, we see losses of 0.00-47.41% in the worse cases. With a maximum oracle, we see losses of 1.42-11.65% for the inlined Rockwell models and 3.64-11.70% for the

Benchmarks models.

In cases where the loss in performance—or gain—are small, one factor that may contribute is the test suite reduction process. Tests are chosen randomly for the reduced suite, based on their ability to cover obligations. In general, efficacy may be essentially identical between the observable and non-observable suites, and poor test cases choices push the average slightly lower—but not in a statistically significant manner. This would explain most of the Rockwell scenarios, as well as Branch-satisfying suites on the Rtp_All system from the Benchmarks dataset. This is a case where Branch and OBranch attain generally the same results—the `if` statements in the model are easily observable—but the average for OBranch is slightly lower due to poor test selection during suite reduction.

Note that performance is never worse for the non-inlined Rockwell models, so the structure of the model may have some impact. This ties into earlier observations on the results. Heavily inlined systems tend to see less improvement from the addition of observability, as there is a shorter path to system output. If there is less potential for improvement in the first place, the additional complexity of observability can downgrade performance if it results in fewer fulfilled obligations.

This can clearly be seen for the Docking_Approach model, which tends to see a large downgrade in performance when Observable Condition Coverage is used to generate suites instead of traditional Condition Coverage. Only 16.98% of the obligations are satisfied for OCondition, rather than 96.04% for Condition. This is also true in MC/DC case, where 35.98% of OMC/DC obligations are covered for for Peterson_All, rather than vs 93.56% for traditional MC/DC—or 56.86% OMC/DC versus 82.00% MC/DC for Car_All. This is also a factor in Decision Coverage for the UMS model, where 3.13% of ODecision obligations are fulfilled, and 92.19% of the obligations are fulfilled for traditional Decision coverage.

If a significantly lower percent of the obligations are fulfilled, the performance will

downgrade. The observable versions of criteria impose much more difficult obligations, so some drop is not surprising. However, if there is a major drop in obligation satisfaction, then it is not unreasonable to also observe some loss in performance. Fortunately, although there are some drawbacks in performance by requiring observability, the *worst* drops are still not major. This is especially true given the dip in satisfaction. In the Docking_Approach example, we lose only 31% in efficacy, despite losing 89% of the obligation satisfaction. This fits with the observation above that observability generally improves results significantly, especially with an output-only oracle.

At first glance, the structure of the model—the level of inlining—appears to have some impact. With the Rockwell models, we never see a downgrade in performance for the non-inlined variants. Similarly—from the Benchmarks examples—Peterson_All, Rtp_All, and Moesi_All are relatively heavily inlined. Docking_Approach and Car_All are more non-inlined. However, those are both cases where the drop is clearly due to low satisfaction of obligations.

However, we do not believe that this is due to inlining alone, but because inlining is a factor informative of the overall *model complexity*. Inlined models tend to see less improvement from observability because they have more complex expressions. Regardless of the length of the path to output, complex expressions suffer more from masking, making it harder to guarantee a clear path to output. In turn, this potentially leads to lower levels of overall satisfaction for the observable variants.

Even though Docking_Approach is not inlined, it does have a deep state space—a series of gated conditions—which results in a longer path to establish to ensure observability. Therefore, we get lower satisfaction of the obligations for the observable variant than the original, which must simply satisfy obligations on individual expressions. The problem, then—inlined or not—is establishing a masking-free path from the expression to the output.

Non-inlined models can offer complex observability requirements because the path length is long—a failure must propagate through a long series of expressions to impact the output. Each individual expression is simple, but there are a large number of them to pass through un-masked. Therefore, the *path length* can be informative of the difficulty of achieving observability—impacting both obligation satisfaction and efficacy. In non-inlined models, the individual statements are simple. This results in trivial satisfaction of traditional coverage criteria, and weaker tests. Even if tests trigger a fault, they tend to be masked on the path to output. As a result, there tends to be a greater performance boost from observability.

Inlined models tend to have a shorter path-to-output, but each expression is much more complex. Therefore, at each expression from activation to output, a failure could be easily masked. As a result, *statement complexity*—which can be judged by the level of inlining—impacts obligation satisfaction as well as the efficacy gap between observable and traditional variants. Suites satisfying the traditional criterion must satisfy much more difficult obligations, and there are fewer opportunities for masking on the path to output. Therefore, the efficacy of the suites satisfying the traditional criterion tend to be relatively effective even without observability. Observability can boost efficacy, but the difficulty of finding a path through the more complex expressions can also cause issues.

The above only discussed combinatorial paths—from expression to output in a single computation cycle. Complexity must also be considered over multiple computation cycles, as observability can be established *after delays*. One additional factor impacting the path to output are the *number of delay expressions*. Failures can be propagated across computation cycles. However, the use of such expressions introduces an additional source of complexity to a model, and test obligations that require a delay observable path can be harder to satisfy.

Factors that can harm efficacy—generally resulting in a reduction in the number of fulfilled obligations—include expression complexity, the length of the combinatorial path from expression to output, and the length of the delayed path from expression to output.

In Table 5.13, we list the maximum improvement seen, and the affected model, for each criterion and oracle. We generally see excellent levels of maximum improvements. With an output-only oracle, we see improvements from 8.11-84.98% for the inlined Rockwell models, 1623.39-1760.52% for the non-inlined Rockwell models, and 353.47-1374.67% for the Benchmark models. As expected, results are more subdued with a maximum oracle. However, we still see improvements of 1.77-181.95% for the inlined Rockwell models, 15.77-31.38% for the non-inlined Rockwell models, and 21.41-230.66% for the Benchmark models.

There are a few observations that can be made from these results. First, the largest improvements tend to be smaller for the MC/DC criterion than for the other criteria. This is reasonable, as the baseline efficacy of MC/DC-satisfying suites tends to be higher to begin with, limiting the potential gains. The largest improvements tend to be for Branch or Condition-satisfying suites. This too is reasonable, as such criteria are weaker to begin with.

Improvements for non-inlined Rockwell models are clearly much greater than the improvements for inlined versions. As discussed above, this can be explained by looking at model complexity. Suites satisfying the traditional criterion must satisfy much more difficult obligations, and there are fewer opportunities for masking on the path to output. Therefore, the efficacy of the suites satisfying the traditional criterion tend to be relatively effective even without observability. For the non-inlined models, the individual expressions are simple—and the traditional criteria can be satisfied relatively trivially. As a result, failures can be more easily masked on the path to

output. However, observability overcomes masking along the path with relative ease. Ultimately, observability produces much more effective test suites for non-inlined models than the traditional criteria.

However, for the Benchmark examples, inlining alone is not always the biggest factor in the max improvement. Almost all of the models that had the maximum improvements from this dataset were relatively inlined. Despite inlining, the level of obligation satisfaction is still high for the observable versions of the criteria. For example, 92.11% versus 96.49% for Moesi_All and 90.07% versus 91.49% for FireFly for OMC/DC and MC/DC. From this, we can see that the complexity of individual statements can limit efficacy—by making it difficult to find masking-clear paths—it does not guarantee weaker results. As long as coverage can be achieved by the test generator, observability seems to have a positive impact. Other factors, like those discussed above, may have more of an impact on satisfaction of obligations than the statement complexity alone.

5.3 Impact of Observability on Test Suite Size and Obligation Satisfaction

Table 5.14: Average Changes in Size of Test Suites and Percentage of Satisfied Obligations

		Size of Test Suites	Obligation Satisfaction
Benchmark	MC/DC	26.45%	-20.44%
	Decision	67.52%	-10.22%
	Condition	55.38%	-18.51%
	Branch	69.68%	-8.02%
Rockwell (Inlined)	MC/DC	25.51%	-4.20%
	Decision	23.02%	-5.72%
	Condition	49.97%	-4.17%
	Branch	7.73%	-3.29%
Rockwell (Non-inlined)	MC/DC	307.88%	-6.49%
	Decision	392.46%	-8.11%
	Condition	376.25%	-6.30%
	Branch	343.54%	-4.82%

In Table 5.14, we present the average change in size of test suites and percentage

of fulfilled obligations when observability is required for each coverage criterion. Regardless the underlying coverage criterion, we see an increase in the number of test cases required for the observable version of the test suite. Fundamentally, observable criteria require more test cases to fulfill their obligations than the traditional variants. Because of the highly specific path to output required for each obligation, there is less overlap between test cases in terms of the obligations satisfied.

We can see that program structure seems to have an impact on the magnitude of the size increase. Moving to an observable criterion results in a massive increase in test suite size for the non-inlined Rockwell models—307.87-392.46%—while there is only a modest increase of 7.73-49.97% for the inlined models. Similarly, on a per-model basis for the Benchmarks examples, many of the models with smaller increases in suite size tend to also be heavily inlined.

This observation makes sense given the discussion above. The obligations for non-observable criteria are formed over individual expressions. If those expressions are simple, the obligations too will be simple. As a result, each test case may cover a variety of obligations with ease. If the model is more heavily inlined, then each obligation will be more complex, and more specialized. There will be less overlap in coverage between test cases [23]. The more heavily inlined the model, the larger the test suite tends to be.

Therefore, model structure has a major impact on the size of the test suite for suites satisfying the *non-observable* criteria. Inlined models start with larger test suites. Then, regardless of the model structure, the addition of observability, increases the size further.

The primary factor influencing the size increase from observability is the length of the path. Each expression encountered along the path imposes additional conditions on maintaining a non-masking path. Therefore, the longer the path length, the more complex the requirements are on the test case. As a result, we see a similar effect

to changing the program structure. The individual test cases are more specialized, and there is less of a chance of overlap in covered obligations. This further explains the larger increase in size for non-inlined models. Non-inlined models have simpler expressions, but much more of them. As a result, the satisfaction complexity is in satisfying path constraints rather than in fulfilling the original expression-level obligations.

The addition of observability results in an increase in the size of test suites. The magnitude of that increase depends on the length of the path from each expression to the output.

Previous work has shown that observability imposes an additional complexity burden on the test case generator, generally resulting in some loss in obligation satisfaction [65, 25]. The results of this study further confirm this. Table 5.14 shows that—on average—there is a loss in obligation satisfaction regardless of the criterion. For the inlined Rockwell models, this average loss ranges from 3.29-5.72%. For the non-inlined versions, this ranges from 4.82-8.11%. Then, for the Benchmarks dataset, the average loss ranges from 8.02-20.44%.

As discussed earlier, a loss in obligation satisfaction is to be expected—the test obligations requires to ensure observability are far more complex than the equivalent obligations when observability is not required. This loss can occur for two reasons. First, if there is no masking-free path, then the obligation will be unfulfillable. This means that observability cannot be established, and thus, a fault in that statement *cannot* influence the output. Generally, this indicates dead code—code that, intentionally or not, cannot affect program output. Occasionally, this is a byproduct of either code reuse—where existing code is reused wholesale—or defensive programming.

However, in some cases, obligations may be too complex for the test generator to fulfill. In such cases, the generator will eventually return an "unknown" verdict. This is an indication that the generator was unable to meet the obligation, and was unable to conclusively determine that it could not be met (the case above). If the obligations are too complex, then the test generator can return weaker test suites because it eventually gives up on finding solutions that fulfill these obligations.

Table 5.15: % of obligations fulfilled, and the % of the unfulfilled obligations that were due to an "unknown" verdict being returned by the test generation for the Benchmarks dataset (Branch and Condition). An NA means there were no obligations for that criterion.

	Branch		**OBranch**		**Condition**		**OCondition**	
	% Fulfilled	**% Unknown**	**% Fulfilled**	**% Unknown**	**% Fulfilled**	**% Unknown**	**% Fulfilled**	**% Unknown**
6counter	NA	NA	NA	NA	100.00%	0.00%	100.00%	0.00%
AlarmFunctionalR2012	92.43%	0.00%	77.55%	22.45%	93.46%	0.00%	80.23%	19.77%
CarAll	100.00%	0.00%	100.00%	0.00%	86.00%	0.00%	65.38%	0.00%
cd	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	88.24%	0.00%
DockingApproach	97.83%	0.00%	36.64%	63.04%	96.04%	0.00%	19.57%	80.37%
DragonAll	99.18%	0.00%	99.18%	0.00%	99.40%	0.00%	99.00%	0.00%
DragonAll2	99.18%	0.00%	99.18%	0.00%	98.73%	0.00%	98.55%	0.00%
durationThm1	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	93.33%	0.00%
ex3	83.33%	0.00%	83.33%	0.00%	78.57%	0.00%	75.00%	0.00%
ex8	83.33%	0.00%	83.33%	0.00%	78.57%	0.00%	75.00%	0.00%
fast_1	100.00%	0.00%	100.00%	0.00%	99.29%	0.00%	93.84%	0.00%
fast_2	100.00%	0.00%	100.00%	0.00%	95.41%	0.00%	91.15%	0.00%
FireFly	97.50%	0.00%	97.50%	0.00%	98.94%	0.00%	98.23%	0.00%
Gas	100.00%	0.00%	8.33%	66.67%	97.06%	0.00%	31.58%	52.63%
HysteresisAll	91.67%	0.00%	91.67%	0.00%	95.00%	5.00%	92.50%	5.00%
IllinoisAll	100.00%	0.00%	100.00%	0.00%	99.38%	0.00%	99.06%	0.00%
Infusion	99.69%	0.00%	36.08%	0.00%	99.54%	0.00%	41.33%	0.00%
MesiAll	93.75%	0.00%	87.50%	0.00%	92.98%	0.00%	59.65%	0.00%
Metros1	100.00%	0.00%	26.67%	0.00%	96.59%	1.14%	30.68%	0.00%
Microwave01	88.11%	3.16%	75.49%	0.00%	93.68%	1.53%	35.90%	0.00%
MoesiAll	0.00%	0.00%	100.00%	0.00%	98.25%	0.00%	95.61%	0.00%
PetersonAll	100.00%	0.00%	100.00%	0.00%	98.48%	0.76%	37.50%	60.42%
ProducerConsumerAll	100.00%	0.00%	81.82%	0.00%	100.00%	0.00%	94.74%	0.00%
ProductionCell	100.00%	0.00%	100.00%	0.00%	98.98%	0.00%	90.00%	0.00%
Readwrit	100.00%	0.00%	100.00%	0.00%	94.36%	3.08%	41.28%	52.05%
RtpAll	98.98%	0.00%	98.98%	0.00%	95.64%	1.66%	27.80%	68.05%
Speed2	83.33%	0.00%	83.33%	0.00%	78.57%	0.00%	75.00%	0.00%
Stalmark	NA	NA	NA	NA	100.00%	0.00%	100.00%	0.00%
SteamBoilerNoArr1	99.47%	0.00%	98.40%	0.00%	99.69%	0.00%	87.46%	0.00%
SteamBoilerNoArr2	100.00%	0.00%	100.00%	0.00%	97.87%	0.00%	97.87%	0.00%
Swimmingpool1	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	98.90%	0.00%
Switch	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%
Switch2	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%
SynapseAll	100.00%	0.00%	100.00%	0.00%	95.95%	0.00%	68.92%	0.00%
Ticket3iAll	98.65%	0.00%	86.49%	0.00%	96.50%	0.00%	31.85%	0.00%
Traffic	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%
Tramway	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	97.62%	0.00%
TwistedCounters	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%
Two Counters	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%
UMS	100.00%	0.00%	NA	NA	95.86%	0.00%	18.62%	0.00%

To better understand the reasons we lose coverage, we have listed the percent of obligations fulfilled and the percent of obligations that resulted in "unknown"

Table 5.16: % of obligations fulfilled, and the % of the unfulfilled obligations that were due to an "unknown" verdict being returned by the test generation for the Benchmarks dataset (Decision and MC/DC).

	Decision		ODecision		MC/DC		OMC/DC	
	% Fulfilled	**% Unknown**	**% Fulfilled**	**% Unknown**	**% Fulfilled**	**% Unknown**	**% Fulfilled**	**% Unknown**
6counter	100.00%	0.00%	100.00%	0.00%	93.33%	0.00%	93.75%	0.00%
AlarmFunctionalR2012	92.27%	0.00%	77.06%	22.94%	92.25%	0.00%	78.02%	21.98%
CarAll	95.00%	0.00%	77.27%	0.00%	82.00%	0.00%	59.62%	0.00%
cd	91.67%	0.00%	75.00%	0.00%	94.12%	0.00%	70.59%	0.00%
DockingApproach	95.08%	0.00%	32.90%	62.18%	91.20%	0.00%	14.84%	85.12%
DragonAll	98.68%	0.00%	98.03%	0.00%	94.00%	0.00%	93.20%	0.00%
DragonAll2	98.05%	0.00%	97.40%	0.00%	89.09%	0.00%	88.55%	0.00%
durationThm1	91.67%	0.00%	83.33%	0.00%	90.00%	0.00%	76.67%	0.00%
ex3	83.33%	0.00%	83.33%	0.00%	71.43%	0.00%	64.29%	0.00%
ex8	91.67%	0.00%	91.67%	0.00%	78.57%	0.00%	71.43%	0.00%
fast_1	96.00%	0.00%	87.50%	0.00%	96.43%	0.00%	90.41%	0.00%
fast_2	92.31%	0.00%	84.88%	0.00%	93.58%	0.00%	87.17%	0.00%
FireFly	97.06%	0.00%	95.10%	0.00%	91.49%	0.00%	90.07%	0.00%
Gas	100.00%	0.00%	16.67%	70.83%	94.12%	2.94%	81.58%	2.63%
HysteresisAll	91.67%	0.00%	91.67%	0.00%	80.00%	5.00%	75.00%	5.00%
IllinoisAll	99.15%	0.00%	98.31%	0.00%	92.81%	0.00%	91.88%	0.00%
Infusion	99.23%	0.00%	39.44%	0.00%	99.08%	0.00%	38.37%	0.00%
MesiAll	93.59%	0.00%	80.77%	0.00%	89.47%	0.00%	56.14%	0.00%
Metros1	94.12%	0.00%	29.41%	0.00%	86.36%	0.00%	23.86%	0.00%
Microwave01	83.07%	3.65%	60.42%	0.00%	75.15%	13.15%	33.79%	0.00%
MoesiAll	97.62%	0.00%	96.43%	0.00%	96.49%	0.00%	92.11%	0.00%
PetersonAll	99.43%	0.00%	97.13%	0.00%	93.56%	0.57%	35.98%	61.55%
ProducerConsumerAll	97.22%	0.00%	86.11%	0.00%	76.32%	0.00%	71.05%	0.00%
ProductionCell	96.88%	0.00%	97.73%	0.00%	86.73%	0.00%	79.09%	0.00%
Readwrit	99.34%	0.00%	96.71%	0.00%	89.74%	3.08%	41.28%	52.05%
RtpAll	98.44%	0.00%	95.31%	0.00%	90.25%	1.66%	27.39%	68.05%
Speed2	83.33%	0.00%	83.33%	0.00%	71.43%	0.00%	64.29%	0.00%
Stalmark	87.50%	0.00%	87.50%	0.00%	41.67%	0.00%	41.67%	0.00%
SteamBoilerNoArr1	99.09%	0.00%	90.00%	0.00%	91.38%	0.00%	78.84%	0.00%
SteamBoilerNoArr2	87.50%	0.00%	87.50%	0.00%	91.49%	0.00%	91.49%	0.00%
Swimmingpool1	100.00%	0.00%	100.00%	0.00%	92.31%	0.00%	91.21%	0.00%
Switch	100.00%	0.00%	100.00%	0.00%	93.33%	0.00%	93.33%	0.00%
Switch2	100.00%	0.00%	100.00%	0.00%	93.33%	0.00%	93.33%	0.00%
SynapseAll	97.92%	0.00%	89.58%	0.00%	90.54%	0.00%	63.51%	0.00%
Ticket3iAll	97.96%	0.00%	84.69%	0.00%	90.13%	0.00%	31.21%	0.00%
Traffic	87.50%	0.00%	75.00%	0.00%	85.00%	0.00%	70.00%	0.00%
Tramway	98.21%	0.00%	96.67%	0.00%	91.80%	0.00%	88.10%	0.00%
TwistedCounters	92.86%	0.00%	92.86%	0.00%	81.82%	0.00%	81.82%	0.00%
Two Counters	87.50%	0.00%	87.50%	0.00%	66.67%	0.00%	66.67%	0.00%
UMS	92.19%	0.00%	2.63%	0.00%	84.96%	0.00%	1.72%	0.00%

verdicts—where the test generator gave up on finding a solution—for each model from the Benchmarks dataset in Tables 5.15-5.16. We do the same for the inlined Rockwell examples in Table 5.17 and the non-inlined variants in Table 5.18. Finally, we list averages for each dataset in Table 5.19.

First, we can see again from Table 5.19 that the observable variants see a lower rate of obligation fulfillment than the traditional criteria. Again, this is expected. In the case of the Rockwell models, we see that there are no situations where the test generator returned an unknown verdict. This means that any loss in such situations is due to *provably* unfulfillable obligations—dead code. This reduction in fulfillment

Table 5.17: % of obligations fulfilled, and the % of the unfulfilled obligations that were due to an "unknown" verdict being returned by the test generation for the Rockwell (inlined) dataset.

	Branch		**OBranch**		**Condition**		**OCondition**	
	% Fulfilled	**% Unknown**	**% Fulfilled**	**% Unknown**	**% Fulfilled**	**% Unknown**	**% Fulfilled**	**% Unknown**
DWM_1	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%
DWM_2	100.00%	0.00%	100.00%	0.00%	99.73%	0.00%	99.73%	0.00%
Microwave	89.41%	0.00%	74.71%	0.00%	94.66%	0.00%	75.64%	0.00%
Latctl_Batch	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	99.23%	0.00%
Vertmax	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%
	Decision		**ODecision**		**MC/DC**		**OMC/DC**	
	% Fulfilled	**% Unknown**	**% Fulfilled**	**% Unknown**	**% Fulfilled**	**% Unknown**	**% Fulfilled**	**% Unknown**
DWM_1	100.00%	0.00%	100.00%	0.00%	98.74%	0.00%	98.74%	0.00%
DWM_2	100.00%	0.00%	100.00%	0.00%	93.93%	0.00%	93.93%	0.00%
Microwave	94.66%	0.00%	69.66%	0.00%	94.02%	0.00%	75.00%	0.00%
Latctl_Batch	100.00%	0.00%	97.83%	0.00%	99.62%	0.00%	98.85%	0.00%
Vertmax	100.00%	0.00%	100.00%	0.00%	96.99%	0.00%	96.99%	0.00%

Table 5.18: % of obligations fulfilled, and the % of the unfulfilled obligations that were due to an "unknown" verdict being returned by the test generation for the Rockwell (non-inlined) dataset.

	Branch		**OBranch**		**Condition**		**OCondition**	
	% Fulfilled	**% Unknown**	**% Fulfilled**	**% Unknown**	**% Fulfilled**	**% Unknown**	**% Fulfilled**	**% Unknown**
DWM_1	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%
DWM_2	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	97.74%	0.00%
Microwave	100.00%	0.00%	75.90%	0.00%	99.15%	0.00%	72.25%	0.00%
Latctl_Batch	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	97.89%	0.00%
Vertmax	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%
	Decision		**ODecision**		**MC/DC**		**OMC/DC**	
	% Fulfilled	**% Unknown**	**% Fulfilled**	**% Unknown**	**% Fulfilled**	**% Unknown**	**% Fulfilled**	**% Unknown**
DWM_1	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	99.90%	0.00%
DWM_2	100.00%	0.00%	94.87%	0.00%	98.68%	0.00%	96.42%	0.00%
Microwave	99.73%	0.00%	66.76%	0.00%	98.73%	0.00%	71.61%	0.00%
Latctl_Batch	100.00%	0.00%	97.64%	0.00%	100.00%	0.00%	97.63%	0.00%
Vertmax	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%	99.77%	0.00%

is acceptable, as such obligations can never be fulfilled.

However, for the complex Benchmarks models, we do see some loss due to the test generator. On average, 4.11% (OBranch), 8.46% (OCondition), 3.90% (ODecision), and 7.41% (MC/DC) of obligations return "unknown" verdicts during test generation. We wish to avoid such situations, as they are situations where we cannot prove that the obligation cannot be fulfilled—the test generator just did not find a solution in time. Some of these obligations may have test cases meeting them. Many will not, but we lack proof in one direction or another.

In the Benchmarks dataset, even the traditional criteria have obligations that result in unknown verdicts—on average, 0.08% (Branch), 0.33% (Condition), 0.09%

Table 5.19: Average % of obligations fulfilled, and the average % of the unfulfilled obligations that were due to an "unknown" verdict being returned by the test generation for each dataset.

	Branch		OBranch		Condition		OCondition	
	% Fulfilled	% Unknown	% Fulfilled	% Unknown	% Fulfilled	% Unknown	% Fulfilled	% Unknown
Benchmarks	94.91%	0.08%	87.88%	4.11%	96.22%	0.33%	75.78%	8.46%
Rockwell (inlined)	97.88%	0.00%	94.94%	0.00%	98.88%	0.00%	94.92%	0.00%
Rockwell (non-inlined)	100.00%	0.00%	95.18%	0.00%	99.83%	0.00%	93.58%	0.00%
	Decision		ODecision		MC/DC		OMC/DC	
	% Fulfilled	% Unknown	% Fulfilled	% Unknown	% Fulfilled	% Unknown	% Fulfilled	% Unknown
Benchmarks	94.60%	0.09%	81.17%	3.90%	86.84%	0.66%	67.08%	7.41%
Rockwell (inlined)	98.93%	0.00%	93.50%	0.00%	96.66%	0.00%	92.70%	0.00%
Rockwell (non-inlined)	99.95%	0.00%	91.85%	0.00%	99.48%	0.00%	93.07%	0.00%

(Decision), and 0.66% (MC/DC) of the obligations time out. However, these percentages are far lower than for the observable variants. This speaks to the complexity of establishing observability, which is often far beyond that of covering the obligations of the host criterion.

The two driving factors in these unknown verdicts are the length of the combinatorial path from expression and output and the number of delay expressions—the length of the delayed path—between the expression and the output. Both increase the complexity of finding a masking-free path between the expression that is the source of the base obligation and an output variable. If the path is more complex, the generator will have a harder time satisfying the test obligations.

Although paths are shorter in inlined models, the individual expressions are more complex than in non-inlined models. Although expressions are simple in non-inlined models, the paths are longer than in inlined models. As a result, the level of inlining does not play a major role in the loss in obligation satisfaction. The level of correlation between inlining and loss in satisfaction is relatively low. The length of the path—whether delayed or immediate—is of far more importance.

The addition of observability results in an decrease in the number of fulfilled obligations. This loss is due to either the discovery of dead code that cannot influence the output or obligations that are too complex for the test generator to

solve.

5.4 The Effect of Observability

For our final research question, we wish to take a look at the effect of observability *itself*. Regardless of the underlying host criterion, does observability have a consistent impact on suite efficacy, oracle sensitivity, structure sensitivity, and obligation fulfillment?

5.4.1 The Choice of Host Criterion

The choice of coverage criterion is often made based on the perceived strength of that criterion. MC/DC is more strenuous to fulfill than Branch Coverage, and therefore, suites satisfying it *should* be more effective. While there are exceptions, this generally bears out in practice. In our study, MC/DC satisfaction results in stronger test suites than Branch Coverage satisfaction.

However, one question we are curious about is—when observability is required, *does the choice of host criterion matter*? Does observability consistently improve results, and is there still reasonable differentiation in the final results to see an impact from the choice of host criterion.

From the results in Tables 5.5 and 5.10, we can still see that the choice of criterion matters. Observability generally results in better test suites, but there is no real consistency in the magnitude of that impact across criteria, oracles, and system structures. The choice of criteria does impact the end result. OMC/DC satisfaction does tend to result in better test suites than OBranch satisfaction. The gap between criteria is often narrower for the observable variants than their traditional variants, but there is still a gap. Therefore, we can conclude that the choice of host criterion still influences the final result.

With traditional coverage criteria, weaker criteria may be chosen for use because they offer *enough* benefit, but are less expensive to fulfill. This is particularly true when test cases are written by human developers. Branch Coverage is easier to understand and explain than MC/DC, and proving that your test cases meet the more strenuous requirements of MC/DC requires more time and effort. If satisfaction of Branch Coverage can be achieved within the time period alloted to testing and offers benefits to the testing process, it may be better to make use of it than to spend the same amount on time attaining partial coverage of MC/DC. Even in the case of automated generation, it may be reasonable to choose to maximize Branch Coverage over attaining partial coverage of MC/DC. If the test generator is unable to satisfy the requirements of MC/DC, then attaining a higher level of Branch Coverage could lead to better suite efficacy.

However, this same trade-off does not necessarily function in an equivalent manner once observability is required. As we can see from the discussion in Section 5.3, the added complexity of observability vastly outweighs the complexity added by the use of a criterion such as MC/DC over Branch Coverage. If the test generation framework employed in this study can satisfy Branch Coverage for a model, it can usually attain similar levels of MC/DC. There is a far more perceptible drop when moving to any of the observable criteria. A gap still exists between Observable Branch and Observable MC/DC, but the leap from non-observable to observable is much greater.

It follows then that—rather than asking which criterion to employ—the more important questions is whether to require observability. In the context of manual test creation, employing observability without tool support is likely to be too expensive to consider in any situation except when safety is absolutely crucial. In the case of automated generation, observability is—at least for the studied programs—reasonable to require. Although there are situations where the loss in coverage due to unknown verdicts is unacceptably high, for most of the studied programs, there were clear

benefits in efficacy.

These results also show that—as long as the test generator can handle the complexity of observability at all—the additional loss from choosing a more complex host criterion is minor. Therefore, we would recommend the use of stronger criteria such as MC/DC over weaker ones when observability can be handled by the test generator. That said, if the combined complexity of observability and criterion is too much for generation to handle, then a tester could first change the host criterion—then drop the observability requirement.

> The choice of host criterion influences the final efficacy, but the largest increase in complexity comes from the addition of observability itself. Varying both dimensions—criterion and observability—may allow testers to find an optimal level of efficacy and complexity.

5.4.2 Oracle Sensitivity

Table 5.20: Average improvement in mutation detection when changing from OO to MX oracle.

	Benchmark	Rockwell (Inlined)	Rockwell (Non-inlined)
OMC/DC	96.77%	10.73%	16.31%
ODecision	93.33%	49.94%	19.38%
OCondition	96.81%	34.53%	16.83%
OBranch	102.69%	59.89%	40.63%
MC/DC	208.76%	13.03%	352.92%
Decision	233.13%	78.20%	384.62%
Condition	297.46%	31.45%	375.41%
Branch	278.98%	97.67%	412.78%

In normal situations, the results of testing are sensitive to the choice of variables monitored as part of the test oracle. We can see this in comparing the results of the maximum and output-only oracles for suites satisfying the traditional non-observable criteria. When results are checked with the maximum oracle, efficacy tends to be much high. This is because masking can prevent program elements from influencing

other variables. With any oracle other than the maximum oracle, suite efficacy depends on the selection of variables monitored by the oracle [24]. This complicates the testing process, as it is not obvious *which* variables should be monitored, and coming up with expected values for any variables other than the output variables can be very difficult.

In theory, observability should be a powerful tool in overcoming oracle sensitivity. By requiring a masking-free path from any targeted expression to the output, we should be able to increase the efficacy of using an output-only oracle. In Table 5.20, we present the average improvement in fault finding when moving from an output-only oracle to the maximum oracle for each coverage criterion, and for each of the three datasets.

From these results, we can see that for the non-inlined Rockwell systems, oracle sensitivity is *greatly* reduced when we require observability—for instance, Branch-satisfying suites improve by 412.78% when changing oracles, but OBranch-satisfying suites only improve by 40.63%. As discussed earlier, non-inlined systems tend to have a large number of simple expressions and long paths to output. These results make sense. The maximum oracle monitors every single expression in the program. Therefore, the size of the maximum oracle is much larger than the output-only oracle, as it is much easier to detect faults. When paired with an output-only oracle, suites satisfying traditional criteria will suffer greatly from masking. Observability overcomes this masking by requiring that each expression be able to influence the output.

We do not see the same magnitude of effect for the aggressively inlined versions of the Rockwell models. Except in the case of Condition Coverage, there is a reduction in oracle sensitivity, but the impact is less. Again, however, these results make intuitive sense. An inlined implementation has fewer expressions. Therefore, the maximum oracle is also smaller—with fewer points of observation. The observable versions of

criteria still produce suites that are less sensitive to the choice of oracle, but there is also potentially less oracle sensitivity to overcome in the first place.

The Benchmark models again fall in between the two extremes. However, we do clearly see the effect of adding observability. The suites satisfying the observable criteria are less sensitive to the choice of oracle than suites satisfying the traditional counterparts. For instance, suites satisfying traditional Branch Coverage improve by 273.10% from the shift in oracle, while suites satisfying OBranch Coverage only improve by 102.51%.

Observability reduces sensitivity to the choice of oracle, by ensuring a masking-free path from expression to the variables monitored by the test oracle.

5.4.3 Structural Sensitivity

Table 5.21: Average change in mutation detection when switching from non-inlined to inlined versions of the Rockwell models.

	Max Oracle	OO Oracle
OMC/DC	-3.32%	1.50%
ODecision	-15.23%	-25.41%
OCondition	-10.71%	-18.56%
OBranch	-16.70%	-20.47%
MC/DC	0.18%	354.62%
Decision	-7.83%	344.09%
Condition	-22.54%	332.22%
Branch	-5.83%	332.24%

Traditional coverage criteria—particularly MC/DC—are known to be sensitive to program structure [23, 50]. With an output-only oracle, suites generated using the inlined version of the program will be far more effective at finding faults than suites generates using the non-inlined version of the program. Because individual expressions are more complex in the inlined program, their test obligations are more complex. There are also, often, fewer opportunities for masking on the path to output, as there are fewer expressions along that path. Observability should help overcome that

sensitivity to structure. Although the individual expressions are simpler, overcoming masking along the path should result in a more robust test suite.

In our experiments, this seems to be the case. Table 5.21 lists the average change in efficacy when switching from non-inlined to inlined version of the Rockwell models—the only models where we can compare multiple structures. On average, we see that—for the traditional suites—there is a major improvement in efficacy when we change program structures. For suites satisfying the observable variants, we actually see a slight downgrade in performance.

For the traditional criteria, if we use a maximum oracle, we see a downgrade in performance when changing program structure instead of the upgrade we saw with an output-only oracle. This is because, with a non-inlined program, the size of the maximum oracle is very large. Each of the many simple expressions is monitored and checked. With an inlined program, the maximum oracle is much smaller—there are fewer expressions. Therefore, with a maximum oracle, changing to an inlined program structure is somewhat detrimental to performance. With traditional criteria—as the maximum oracle is generally prohibitively expensive—we would recommend inlining code to improve the performance of the output-only oracle.

The above results make sense then as, with observability, we essentially see the same effect. There is no benefit from changing program structure, as the increased complexity of individual statements is replicated in the masking-free path to output required to attain observability. Instead, there is a slight downgrade in performance because the individual statements are more complex. When observability is required, a simpler program structure may be slightly preferable.

Observability reduces sensitivity to the program structure by capturing the complexity benefits of inlining in the path from expression to output.

5.5 Threats to Validity

External Validity: Our study has focused on a relatively small number of systems but, nevertheless, we believe the systems are representative of the critical systems domain, and our results are generalizable to other systems in that domain.

We have used one method of test generation—counterexample-based generation. There are many methods of generating tests and these methods may yield different results. Counterexample-based testing is used to produce coverage-directed test cases because it is a method used widely in testing safety-critical systems.

For each model and criteria, we have built 50 reduced test suites reduced using a simple greedy algorithm. It is possible that larger sample sizes may yield different results. However, in previous studies, smaller numbers of reduced test suites have been seen to produce consistent results [50].

Construct Validity: In our study, we primarily measure fault finding over seeded faults, rather than real faults encountered during development. However, Andrews et al. showed that seeded faults lead to similar conclusions to those obtained using real faults [4] for the purpose of measuring test effectiveness and Just et al. found a positive correlation between mutant detection and fault detection [37]. We have assumed these conclusions hold true in our domain/language, where examples of real faults are rare.

To control experiment costs, we limited the number of mutants used per model to 500. When more than 500 mutants exist, a random selection was used to avoid bias in mutant selection. While the selection of specific mutants is randomized, the distribution is matched to the full distribution of possible mutants in the model. In our experience, mutants sets greater than 100 result in very similar fault finding; we generated up to 500 to further increase our confidence that no bias was introduced.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions beyond these analyses are met, and have favored non-

parametric methods. In cases in which the base assumptions are clearly not met, we have avoided using statistical methods. Notably, we have avoided statistical inference across case examples.

Chapter 6

Relate Work

In this chapter, we will discuss our prior work on observability, the role of adequacy criteria in test case generation, other notions of observability, and other topics related to this work.

6.1 Prior Work on Observability

This work is an extension of our prior work defining and exploring the concept of observability [65, 25, 67]. We first proposed the concept of observability as an extension of the MC/DC coverage criterion [65]. An extended study found that that OMC/DC was more effective—and overcame many of the weaknesses of—traditional coverage criteria [25].

In a recent study, we extended the original tagging semantics of MC/DC in order to generate path conditions as part of Dynamic Symbolic Execution [67]. This work used OMC/DC purely as a test generation target rather than a general adequacy measurement approach. A source of optimistic inaccuracy in the original definition of OMC/DC was addressed by requiring value inequality of expressions from two branches when propagating `if` conditions. This approach was also able to explicitly terminate when there is no feasible paths. In the regular model-based test generation approach used in this and the other past work, a timeout is usually estimated and manually set in order to terminate the generation process. The DSE-based approach, as a result, could complete generation in a more efficient manner.

This work extends previous efforts by decoupling the notion of observability from

MC/DC and exploring its application as a generic addition to any coverage criterion. While we found that MC/DC was still the most effective host criterion in many applications, this was not a universal case. This decoupling allows us to explore the impact of choosing a host criterion and to explore the efficacy of observability as a general construct of adequacy criteria. Our experimental work also considers a far wider range of programs than previously explored in order to better understand the general efficacy of observability-based coverage criteria.

6.2 Adequacy Criteria Efficacy in Test Generation

Automated test generation relies on the selection of a measurable test goal. Adequacy criteria, such as the coverage criteria that are the focus of this study are commonly used for this purpose. However, coverage is merely an approximation of a harder to quantify goal—"finding faults". The need to rely on approximations leads to two questions that researchers have examined multiple times. First, *do such proxies produce effective tests?* If so, *which criteria should be used to generate tests?*

Answers to these two questions are—to date—inconclusive. Some studies have noted positive correlation between coverage level and fault detection [21, 47, 46], while other work paints a negative portrait of coverage [33]. Our prior work in search-based test generation for Java programs has found that coverage level is more strongly indicative of efficacy than factors such as suite size [21]. However, in our prior studies of model-based generation, tests generated specifically to achieve coverage were often outperformed by randomly-generated tests [25, 22, 62].

Results to date are promising, given the complexity of some of the faults detected [58, 2, 21, 3]. However, automated generation does not yet produce human competitive results [17]. Ultimately, if automated generation is to have an impact on testing practice, it must produce results that match—or, ideally, outperform—manual testing efforts. The efficacy of suites generated for many coverage criteria is

limited by issues such as masking. Choices about how code is written [23, 50] and the selection of test oracle [24, 60, 61] impact the efficacy of some criteria. The notion of observability was designed to address both issues.

6.3 Coverage Criteria in Lustre and Function Block Diagram

Lustre and Function Block Diagram (FBD) are data-flow languages that describe how inputs are transformed into outputs instead of describing the control flow of the program. Researchers studying coverage criteria for Lustre [39] and FBD [34] implicitly investigated observability by examining variable propagation from the inputs to the outputs.

Some of the structural coverage criteria proposed specifically for Lustre are based on *activation conditions* that are defined as the condition upon which a data flow is transferred from the input to the output of a path. When the activation condition of a path is true, any change in input causes modification of the output within a finite number of steps [39]. Coverage metrics for FBD are based on a *d-path condition* that is similar to activation conditions in Lustre [34].

These coverage criteria in Lustre and FBD are different from the notion of observability in several respects. First, these metrics check if specific inputs affect the outputs and measure the coverage of variable propagation on all possible paths. Observability, on the other hand, checks if each test obligation from the host criterion affects the monitored variables, and determines if a path exists which propagates the effect of the obligation. Second, observability requires a stronger notion of how a decision must be exercised.

6.4 Observability in Hardware Testing

Observability has been studied in testing of hardware logic circuits. Observability-based code coverage metric (OCCOM) is a technique where tags are attached to

internal states in a circuit and the propagation of tags is used to predict the actual propagation of errors (corrupted state) [10, 13]. A variable is tagged when there is a possible change in the value of the variable due to an fault. The observability coverage can be used to determine whether erroneous effects that are activated by the inputs can be observed at the outputs.

The key differences between our notion of observability and OCCOM are twofold: (1) our notion of observability investigates variable value propagation, while OCCOM investigates fault propagation and (2) OCCOM has pessimistic inaccuracy because of tag cancellation. When both positive and negative tags exist in the same assignment (e.g., different tags in an ADDER or the same tags in a COMPARATOR cancel each other out), no tag is assigned [10] or an unknown tag "?"[13] is used. Variables without tags or with unknown tags are not considered to carry an observable error.

In this work—since we do not make a distinction between positive and negative tags—we do not have tag cancellation or the corresponding pessimistic inaccuracy. Extended work in [12] may fix pessimistic inaccuracy by producing test vectors with specific values, but is highly infeasible.

6.5 Strong Mutation Coverage

Mutants are copies of programs where synthetic faults are inserted either through automated code transformation or by hand [37, 4]. Mutants are a common method of test suite evaluation in test generation research when real faults are not available for a system. The mutations introduced generally match one or more models of the types of mistakes that real developers make when building code. Generally, mutants are introduced with the intent that they not be trivially detected—they are both syntactically and semantically valid [49]. That is, the mutants will compile, and no mutant will "crash" the system. Detection of mutants has also been the basis of multiple adequacy criteria [16, 36]. In theory, if a suite detects more mutants, it will

also be more adequate at fault detection.

Commonly, the concept of mutation detection is measured in two ways. In *weak* mutation coverage, a mutant is considered detected if the mutated statement is reached, and the value of *that expression* is corrupted [40]. In *strong* mutation coverage, the resulting corruption must influence an output variable. This is a direct parallel to traditional and observable coverage criteria. In traditional criteria, the targeted statement must execute in a desired manner. In observable criteria, the exercised statement must influence the output. If strong mutation coverage is to be achieved, then we must have observability on the mutated statement.

Strong mutation coverage is very difficult to ensure, and expensive to measure [16]. Therefore, weak mutation coverage is often used instead, as a high level of weak mutation coverage can be more easily reached. Observability, as proposed in this work, offers a means to increase strong mutation coverage of faults in Boolean decisions. Although this is only a subset of all possible mutations, the results of our experiments do show that observability increases the strong mutation coverage of test suites.

6.6 Dynamic Taint Analysis

Dynamic taint analysis, or dynamic information flow analysis, marks and tracks data in a program at runtime, similar to our tagging semantics. This technique has been used in security as well as software testing and debugging [41, 9]. Taint propagation occurs in both explicit information flow (i.e., data dependencies) and implicit information flow (control dependencies). Although the way in which markings are combined varies based on the application, the default behavior is to union them [9]. Thus, dynamic taint analysis is conservative and does not consider masking. More accurate techniques for information flow modeling, such as [64], define path conditions quite similar to those used in this paper to prove *non-interference*, that is, the non-observability of a variable or expression on a particular output.

6.7 Dynamic Program Slicing

Dynamic program slicing [1] computes a set of statements that influence the variables used at a program point for a particular execution. This can identify all variables that contribute to a specific program point, including output. However, similarly to dynamic taint analysis, it does not consider masking. Checked coverage uses dynamic slicing to assess oracle quality, where oracles are program assertions [57]. Given a test suite, it yields a percentage of all statements that contribute to the value of any assertion (i.e., are observable at that assertion) vs. the total number of statements covered by the test suite. This work is designed to assess the *oracle*, not the test suite.

Chapter 7

Conclusions and Future Work

Many test adequacy criteria are highly sensitive to how statements are structured or the choice of test oracle. This sensitivity is caused by the fact that the obligations for structural coverage criteria are only posed over specific syntactic elements—statements, branches, conditions. Such obligations ensure that execution *reaches* the element of interest, and exercises it in the prescribed manner. However, no constraints are imposed on the execution path *after* this point. We are not guaranteed to observe a failure just because a fault is triggered.

To address this issue, we have proposed the concept of *observability*—an extension to coverage criteria based on Boolean expressions that has the potential to eliminate masking. Observable coverage criteria combine the test obligations of their host criterion with an additional path condition that increases the likelihood that a fault encountered when executing the element of interest will propagate to a variable monitored by the test oracle. We hypothesize that this additional observability constraint will improve the effectiveness of the host criterion—no matter which criterion is chosen—particularly when used as a test generation target, paired with common output-based test oracles.

Our study, conducted over five industrial systems from Rockwell Collins and an additional forty open-source systems, has revealed that test suites satisfying Observable MC/DC are generally the most effective, killing 95.61% of mutants on average (MX oracle) and 87.03% (OO oracle) for the inlined Rockwell models, 98.85% (MX)/85.88% (OO) for the non-inlined Rockwell models, and 89.62% (MX)/65.14%

(OO) for the Benchmarks models. Overall, we found that adding observability tends to improve efficacy over satisfaction of the traditional criteria, with average improvements of up to 392.44% in mutation detection and per-model improvements of up to 1654.38%.

Some of the factors that can harm efficacy—generally resulting in a reduction in the number of fulfilled obligations—include expression complexity, the length of the combinatorial path from expression to output, and the length of the delayed path from expression to output. The addition of observability results in an increase in the size of test suites. The magnitude of that increase depends on the length of the path from each expression to the output. In addition, the addition of observability results in an decrease in the number of fulfilled obligations. This loss is due to either the discovery of dead code that cannot influence the output or obligations that are too complex for the test generator to solve. The choice of host criterion influences the final efficacy, but the largest increase in complexity comes from the addition of observability itself. Varying both dimensions—criterion and observability—may allow testers to find an optimal level of efficacy and complexity. Ultimately, our hypothesis has proven accurate—observability reduces sensitivity to the choice of oracle and to the program structure.

Based on our results, *observability* is a valuable extension—regardless of the chosen host criterion. The addition of observability increases test efficacy and produces test suites that are robust to changes in the structure of program or the variables under monitored by test oracle. While our results are encouraging, there are areas open for exploration in future research:

- **Extension to other coverage criteria:** A variety of coverage criteria have been proposed for logical expressions, some potentially more effective than MC/DC [68]. We will explore the effect of extending such criteria to offer observability.

- **Oracle data selection:** We used two types of oracles representing different extremes. Maximum oracles monitor all internal and output variables, and output-only oracles monitor only the output variables. However, we have found that some level of oracle sensitivity could be overcome with intelligently constructed oracles [24]. We intend to further consider whether such oracles could be more effective in situations where observability constraints are too difficult for the test generator.

- **Selection of solver used for test generation:** While conducting out study, we found that the model checker had difficulties with satisfying the observability constraints for some models. Further, we witnessed varying efficacy performance between the underlying solvers powering out employed test generation approach. We will extend our work in the future to quantify and further explore the choice of solver and its effect on suite efficacy.

- **Method of test generation:** In this work, we have used model-based test generation. In past work, we also used Dynamic Symbolic Execution to generate test suites satisfying Observable MC/DC [67]. In the future, we would like to explore other methods of generating tests for observable criteria, such as search-based generation.

References

[1] H. Agrawal and J.R. Horgan. "Dynamic program slicing". In: *Proceedings of the ACM SIGPLAN 1990 Conf. on Programming Language Design and Implementation*. Vol. 25. 1990, pp. 246–256.

[2] M. Moein Almasi et al. "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application". In: *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE) — Software Engineering in Practice Track (SEIP)*. ICSE 2017. Buenos Aires, Argentina: ACM, 2017.

[3] Hussein Almulla, Alireza Salahirad, and Gregory Gay. "Using Search-Based Test Generation to Discover Real Faults in Guava". In: *Proceedings of the Symposium on Search-Based Software Engineering*. SSBSE 2017. Paderborn, Germany: Springer Verlag, 2017.

[4] J.H. Andrews et al. "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria". In: *IEEE Transactions on Software Engineering* 32.8 (Aug. 2006), pp. 608–624. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.83.

[5] Andrea Arcuri. "It really does matter how you normalize the branch distance in search-based software testing". In: *Software Testing, Verification and Reliability* 23.2 (2013), pp. 119–147.

[6] E.T. Barr et al. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* 41.5 (May 2015), pp. 507–525. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2372785.

[7] J. Chilenski. *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*. Tech. rep. DOT/FAA/AR-01/18. Washington, D.C.: Office of Aviation Research, Apr. 2001.

[8] J. J. Chilenski and S. P. Miller. "Applicability of Modified Condition/Decision Coverage to Software Testing". In: *Software Engineering Journal* (Sept. 1994), pp. 193–200.

[9] J. Clause, W. Li, and A. Orso. "Dytan: a generic dynamic taint analysis framework". In: *Proceedings of the 2007 Int'l Symposium on Software Testing and Analysis*. 2007, pp. 196–206.

[10] S. Devadas, A. Ghosh, and K. Keutzer. "An observability-based code coverage metric for functional simulation". In: *Proceedings of the 1996 IEEE/ACM Int'l Conf. on Computer-Aided Design*. 1996, pp. 418–425.

[11] Esterel-Technologies. *SCADE Suite Product Description*. http://www.esterel-technologies.com/v2/ scadeSuiteForSafetyCriticalSoftwareDevelopment/ index.html. 2004.

[12] F. Fallah, P. Ashar, and S. Devadas. "Functional vector generation for sequential HDL models under an observability-based code coverage metric". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 10.6 (2002), pp. 919–923.

[13] F. Fallah, S. Devadas, and K. Keutzer. "OCCOM-efficient computation of observability-based code coverage metrics for functional verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.8 (2001), pp. 1003–1015.

[14] Matthias Felleisen and Robert Hieb. "The revised report on the syntactic theories of sequential control and state". In: *Theor. Comput. Sci.* 103.2 (Sept. 1992), pp. 235–271. ISSN: 0304-3975.

[15] G. Fraser and A. Arcuri. "Whole Test Suite Generation". In: *Software Engineering, IEEE Transactions on* 39.2 (Feb. 2013), pp. 276–291. ISSN: 0098-5589. DOI: 10.1109/TSE.2012.14.

[16] Gordon Fraser and Andrea Arcuri. "Achieving scalable mutation-based generation of whole test suites". In: *Empirical Software Engineering* 20.3 (2014), pp. 783–812.

[17] Gordon Fraser et al. "Does Automated White-box Test Generation Really Help Software Testers?" In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA. Lugano, Switzerland: ACM, 2013, pp. 291–301. ISBN: 978-1-4503-2159-4. DOI: 10.1145/2483760.2483774. URL: http://doi.acm.org/10.1145/2483760.2483774.

[18] Andrew Gacek. *JKind - a Java implementation of the KIND model checker*. https://github.com/agacek. 2015.

[19] A. Gargantini and C. Heitmeyer. "Using Model Checking to Generate Tests from Requirements Specifications". In: *Software Engineering Notes* 24.6 (Nov. 1999), pp. 146–162.

[20] G. Gay, S. Rayadurgam, and M. P. E. Heimdahl. "Automated Steering of Model-Based Test Oracles to Admit Real Program Behaviors". In: *IEEE Transactions on Software Engineering* 43.6 (June 2017), pp. 531–555. ISSN: 0098-5589. DOI: 10.1109/TSE.2016.2615311.

[21] Gregory Gay. "The Fitness Function for the Job: Search-Based Generation of Test Suites that Detect Real Faults". In: *Proceedings of the International Conference on Software Testing*. ICST 2017. Tokyo, Japan: IEEE, 2017.

[22] Gregory Gay et al. "Moving the Goalposts: Coverage Satisfaction is Not Enough". In: *Proceedings of the 7th International Workshop on Search-Based Software Testing*. SBST 2014. Hyderabad, India: ACM, 2014, pp. 19–22. ISBN: 978-1-

4503-2852-4. DOI: 10.1145/2593833.2593837. URL: http://doi.acm.org/10.1145/2593833.2593837.

[23] Gregory Gay et al. "The Effect of Program and Model Structure on the Effectiveness of MC/DC Test Adequacy Coverage". In: *ACM Trans. Softw. Eng. Methodol.* 25.3 (July 2016), 25:1–25:34. ISSN: 1049-331X. DOI: 10.1145/2934672. URL: http://doi.acm.org/10.1145/2934672.

[24] G. Gay et al. "Automated Oracle Data Selection Support". In: *Software Engineering, IEEE Transactions on* PP.99 (2015), pp. 1–1. ISSN: 0098-5589. DOI: 10.1109/TSE.2015.2436920.

[25] G. Gay et al. "The Risks of Coverage-Directed Test Case Generation". In: *Software Engineering, IEEE Transactions on* PP.99 (2015). ISSN: 0098-5589. DOI: 10.1109/TSE.2015.2421011.

[26] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. "Coverage and Its Discontents". In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward!'14. Portland, Oregon, USA: ACM, 2014, pp. 255–268. ISBN: 978-1-4503-3210-1. DOI: 10.1145/2661136.2661157. URL: http://doi.acm.org/10.1145/2661136.2661157.

[27] G. Hagen. "Verifying safety properties of Lustre programs: an SMT-based approach". PhD thesis. University of Iowa, Dec. 2008.

[28] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Klower Academic Press, 1993.

[29] N. Halbwachs et al. "The Synchronous Dataflow Programming Language Lustre". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320.

[30] Mats P.E. Heimdahl and George Devaraj. "Test-suite Reduction for Model Based Tests: Effects on test quality and Implications for Testing". In: *Proc. of the 19th IEEE Int'l Conf. on Automated Software Engineering (ASE)*. Linz, Austria, Sept. 2004.

[31] M.P.E. Heimdahl, G Devaraj, and R.J. Weber. "Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria?" In: *Proc. of the Eighth IEEE Int'l Symp. on High Assurance Systems Engineering (HASE)*. Tampa, Florida, Mar. 2004.

[32] W.E. Howden. "Theoretical and Empirical Studies of Program Testing". In: *IEEE Transactions on Software Engineering* 4.4 (1978), pp. 293–298. ISSN: 0098-5589. DOI: http://doi.ieeecomputersociety.org/10.1109/TSE.1978.231514.

[33] Laura Inozemtseva and Reid Holmes. "Coverage is Not Strongly Correlated with Test Suite Effectiveness". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 435–445. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568271. URL: http://doi.acm.org/10.1145/2568225.2568271.

[34] E. Jee et al. "A data flow-based structural testing technique for FBD programs". In: *Information and Software Technology* 51.7 (2009), pp. 1131–1139.

[35] N. Juristo, A.M. Moreno, and S. Vegas. "Reviewing 25 years of Testing Technique Experiments". In: *Empirical Software Engineering* 9.1 (2004), pp. 7–44.

[36] Rene Just, Franz Schweiggert, and Gregory M. Kapfhammer. "MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler". In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. Washington, DC, USA: IEEE Computer

Society, 2011, pp. 612–615. ISBN: 978-1-4577-1638-6. DOI: 10.1109/ASE.2011.6100138. URL: http://dx.doi.org/10.1109/ASE.2011.6100138.

[37] René Just et al. "Are mutants a valid substitute for real faults in software testing?" In: *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*. Hong Kong, Nov. 2014, pp. 654–665.

[38] Edward Kit and Susannah Finzi. *Software Testing in the Real World: Improving the Process*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995. ISBN: 0-201-87756-2.

[39] A. Lakehal and I. Parissis. "Structural test coverage criteria for Lustre programs". In: *Proceedings of the 10th Int'l workshop on Formal methods for industrial critical systems*. 2005, pp. 35–43.

[40] Brian Marick. "The Weak Mutation Hypothesis". In: *Proceedings of the Symposium on Testing, Analysis, and Verification*. TAV4. Victoria, British Columbia, Canada: ACM, 1991, pp. 190–199. ISBN: 0-89791-449-X. DOI: 10.1145/120807.120825. URL: http://doi.acm.org/10.1145/120807.120825.

[41] W. Masri, A. Podgurski, and D. Leon. "Detecting and debugging insecure information flows". In: *Proceedings of the 15th Int'l Symposium on Software Reliability Engineering*. 2004, pp. 198–209.

[42] *Mathworks Inc. Simulink*. http://www.mathworks.com/products/simulink. 2015.

[43] *MathWorks Inc. Stateflow*. http://www.mathworks.com/stateflow. 2015.

[44] Phil McMinn. "Search-based Software Test Data Generation: A Survey". In: *Software Testing, Verification and Reliability* 14 (2004), pp. 105–156.

[45] Phil McMinn, Mark Stevenson, and Mark Harman. "Reducing Qualitative Human Oracle Costs Associated with Automatically Generated Test Data". In: *Proceedings of the First International Workshop on Software Test Output Validation*. STOV '10. Trento, Italy: ACM, 2010, pp. 1–4. ISBN: 978-1-4503-0138-1. DOI: 10.1145/1868048.1868049. URL: http://doi.acm.org/10.1145/1868048.1868049.

[46] A. Mockus, N. Nagappan, and T.T. Dinh-Trong. "Test coverage and post-verification defects: A multiple case study". In: *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*. Oct. 2009, pp. 291–301. DOI: 10.1109/ESEM.2009.5315981.

[47] A.S. Namin and J.H. Andrews. *The influence of size and coverage on test suite effectiveness*. ACM, 2009.

[48] William Perry. *Effective Methods for Software Testing, Third Edition*. New York, NY, USA: John Wiley & Sons, Inc., 2006. ISBN: 9780764598371.

[49] M. Pezze and M. Young. *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, Oct. 2006.

[50] A. Rajan, M.W. Whalen, and M.P.E. Heimdahl. "The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage". In: *Proceedings of the 30th International Conference on Software Engineering*. ACM. 2008, pp. 161–170.

[51] A. Rajan et al. *Requirements Coverage as an Adequacy Measure for Conformance Testing*. Springer, 2008.

[52] S. Rayadurgam and M.P.E. Heimdahl. "Coverage Based Test-Case Generation Using Model Checkers". In: *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*. IEEE Computer Society, Apr. 2001, pp. 83–91.

[53] Grigore Rosu and Traian Florin Serbanuta. "An overview of the K semantic framework". In: *J. of Logic and Algebraic Programming* 79.6 (2010), pp. 397–434. ISSN: 1567-8326. DOI: 10.1016/j.jlap.2010.03.012.

[54] RTCA. *DO-178B: Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.

[55] RTCA/DO-178C. *Software Considerations in Airborne Systems and Equipment Certification*.

[56] S.A. Vilkomir and J.P. Bowen. "Reinforced condition/decision coverage (RC/DC): A new criterion for software testing". In: *Lecture Notes in Computer Science* 2272 (2002), pp. 291–308.

[57] D. Schuler and A. Zeller. "Assessing Oracle Quality with Checked Coverage". In: *Proceedings of the Fourth IEEE Int'l Conf. on Software Testing, Verification and Validation*. 2011, pp. 90–99.

[58] Sina Shamshiri et al. "Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges". In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ASE 2015. Lincoln, NE, USA: ACM, 2015.

[59] M. Staats, G. Gay, and M.P.E. Heimdahl. "Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing". In: *Proceedings of the 2012 Int'l Conf. on Software Engineering*. IEEE Press. 2012, pp. 870–880.

[60] M. Staats, G. Gay, and M.P.E. Heimdahl. "Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing". In: *Proceedings of the 2012 International Conference on Software Engineering*. Zurich, Switzerland, 2012, pp. 870–880. ISBN: 978-1-4673-1067-3. URL: http://dl.acm.org/citation.cfm?id=2337223.2337326.

[61] M. Staats, M.W. Whalen, and M.P.E. Heimdahl. "Better testing through oracle selection (NIER track)". In: *Proceedings of the 33rd Int'l Conf. on Software Engineering*. 2011, pp. 892–895.

[62] Matt Staats et al. "On the Danger of Coverage Directed Test Case Generation". English. In: *Fundamental Approaches to Software Engineering*. Ed. by Juan de Lara and Andrea Zisman. Vol. 7212. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 409–424. ISBN: 978-3-642-28871-5. DOI: 10.1007/978-3-642-28872-2_28. URL: http://dx.doi.org/10.1007/978-3-642-28872-2_28.

[63] E.J. Weyuker. "The oracle assumption of program testing". In: *13th International Conference on System Sciences*. 1980, pp. 44–49.

[64] Michael W. Whalen, David A. Greve, and Lucas G. Wagner. "Model Checking Information Flow". In: *David Hardin, editor, Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer-Verlag, Berlin Germany, Mar. 2010.

[65] M. Whalen et al. "Observable Modified Condition/Decision Coverage". In: *Proceedings of the 2013 Int'l Conf. on Software Engineering*. ACM, May 2013.

[66] Frank Wilcoxon. "Individual Comparisons by Ranking Methods". English. In: *Biometrics Bulletin* 1.6 (1945), ISSN: 00994987. URL: http://www.jstor.org/stable/3001968.

[67] D. You et al. "Efficient observability-based test generation by dynamic symbolic execution". In: *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. Nov. 2015, pp. 228–238. DOI: 10.1109/ISSRE.2015.7381816.

[68] Y.T. Yu and M.F. Lau. "A Comparison of MC/DC, MUMCUT and Several Other Coverage Criteria for Logical Decisions". In: *Journal of Systems and Software* 79.5 (2006), pp. 577–590. ISSN: 0164-1212.